universität innsbruck

Increasing throughput of server applications by using asynchronous techniques: A case study on CoAP.NET

Philip Wille

Bachelor Thesis

Supervisor:
Dr. Michael Felderer
Department of Computer Science
Universität Innsbruck

Innsbruck, 14. November 2021

Inhaltsverzeichnis

1 Einleitung

Heutzutage haben Webdienste einen hohen Stellenwert im Internet. Sie sind häufig ein essenzieller Bestandteil von Anwendungen und werden durch eine *Representation State Transfer* (REST)-Schnittstelle für Anwender und Entwickler angeboten. Auch das Internet of Things (IoT) bekommt mit fortschreitender Zeit eine wichtigere Rolle in der Entwicklung von Anwendungen, wie zum Beispiel Hausautomatisierung oder smarte Energieverwaltung.

Um die REST-Architektur auch für eingeschränkte Geräte anbieten zu können, wurde mit *Constrained RESTful Environments* (CoRE) eine geeignete Form gebildet. Somit können solche Geräte, wie zum Beispiel 8-Bit-Mikrocontroller mit begrenztem Arbeitsspeicher und Readonly-Memory (ROM), und Netzwerke, wie zum Beispiel *IPv6 over Low-Power Wireless Area Networks (6LoWPANs)*, solche Architekturen verwenden und realisieren. Damit eine Fragmentierung von Nachrichten in solchen Netzwerken so wenig wie möglich auftritt, wird im Constrained Application Protocol (CoAP) ein so niedriger Nachrichten-Overhead wie möglich angestrebt.

Mit CoAP wurde die Entwicklung eines generischen Webprotokolls für die speziellen Anforderungen dieser eingeschränkten Umgebungen, insbesondere mit Hauptaugenmerk auf Energie-, Gebäudeautomatisierungs- und andere Machine-to-Machine (M2M) Anwendungen angestrebt. Dabei sollte CoAP keine komprimierte Abwandlung von HTTP sein, sondern vielmehr eine Submenge von HTTP mit Verwendung von REST, die für M2M-Szenarien optimiert ist. Somit könnte CoAP leicht dazu verwendet werden, einfache HTTP-Schnittstelle in ein kompaktes Protokoll überzuführen, jedoch bietet CoAP Funktionen die speziell für Machine-to-Machine Anwendungen Verwendung finden. Diese Funktionen sind:

- Eingebaute Entdeckung von, im Netzwerk angebotenen Services und Ressourcen.
- Mutlicast-Unterstützung.
- Asynchronen Nachrichtenaustausch.

1.1 Constrained Application Protocol (CoAP)

Das Constrained Application Protocol, abgekürzt CoAP, ist ein Internetprotokoll, das speziell für M2M Anwendungen, wie zum Beispiel smarte Energieverwaltung oder Hausautomatisierung (Internet of Things (IoT)), entwickelt wurde. Dabei bietet das Protokoll ein REST-ähnliches Interface für Mikrocontroller mit angeschlossenen Aktoren oder Sensoren (*constrained nodes*) oder auch drahtlose Sensornetze (*constrained networks*) an. Die sogenannten *nodes* besitzen meist einen angeschlossenen 8-Bit-Mikrocontroller, der nur eine kleine begrenzte Menge an Arbeitsspeicher (RAM) und Readonly-Memory (ROM) beinhaltet. Bei *contstrained networks*, wie z.B. *IPv6 over Low-Power Wireless*

Area Networks (6LoWPANs), spielt die hohe Paketverlustrate und der für solche Netzwerke typische Datendurchsatz von wenigen 10 kbit/s eine große Rolle. CoAP ist durch den RFC 7252 von Shelby, Hartke und Bormann [3] spezifiziert.

Dabei bietet das *Constrained Application Protocol* ein Interaktionsmodell für Anfragen und Antworten zwischen den, in der Anwendung definierten Endpunkten an. Auch ist eine eingebaute Entdeckung von im Netzwerk angebotenen Services und Ressourcen im Protokoll definiert. Zusätzlich werden Hauptbestandteile des Internets, wie *Unique Resource Identifiers* (URIs) und *Internet Media Types* (zum Beispiel *application/json*) angeboten.

Eine Unterstützung von *Multicast* ist gegeben, wie auch ein sehr geringer Mehraufwand in der Datenübertragung. Dabei wurde Wert darauf gelegt, es einfach für eingebettete Systeme zu halten.

Zur Datenübertragung nutzt CoAP das User Datagram Protocol (UDP). Dieses unterscheidet sich zu Transmission Control Protocol (TCP) in den folgenden Punkten:

- kein Sitzungsaufbau von Sender zu Empfänger (Handshake).
- Stellt nicht sicher, ob alle Pakete beim Empfänger eintreffen.
- Geht ein Paket verloren, wird dieses nicht erneut versendet.

Die folgenden Funktionen sind ein essenzieller Bestandteil von CoAP:

- Erfüllung von M2M Anforderungen in eingeschränkten Umgebungen.
- UDP-Verbindung mit optionaler Zuverlässigkeit, die Unicast- und Multicast Anfragen unterstützt.
- Asynchroner Nachrichtenaustausch.
- Niedriger Mehraufwand durch veränderten Header und niedriger Komplexität des Parsings.
- URI- und Content-Typen-Support.
- Einfache Proxy- und Caching-Unterstützung.
- Zustandsloses HTTP-Mapping, das die Entwicklung von Proxys erlaubt, die den Zugriff auf CoAP Ressourcen über HTTP auf einheitliche Weise ermöglichen, oder für einfache HTTP-Schnittstellen, die alternativ über CoAP realisiert werden können.
- Sicherheitsmechanismen durch das Anbinden von *Datagram Transport Layer Security* (DTLS).

Begriffe in CoAP

Um den Kontext innerhalb des *Constrained Application Protocols* zu verstehen, werden nachfolgend die wichtigsten Begriffe in CoAP kurz erklärt:

- Endpunkt (Endpoint):
  - Ein Endpunkt lebt auf einen "Knoten". Ein Knoten ist vergleichbar mit dem Begriff "Host", der vorwiegend in Internetstandards Erwähnung findet.
  - Ein Endpunkt wird durch Multiplexing-Informationen auf der Transportschicht identifiziert, die eine UDP-Portnummer und eine Sicherheitszuordnung enthalten könnte.
- Ursprungsserver (Origin Server):
  - Der Server, auf dem sich eine bestimmte Ressource befindet oder erstellt werden soll.
- Bestätigende Nachricht (Confirmable Message):
  - Einige Nachrichten benötigen eine Bestätigung des Empfängers. Diese Nachrichten werden als *bestätigt* behandelt.
  - Falls keine Pakete während der Übertragung verloren gingen, wird für jede Nachricht, die bestätigt werden muss, exakt eine Nachricht des Typs *Acknowledgement* oder *Reset* an den Sender zurückgesendet.
- Nicht bestätigende Nachricht (Non-confirmable Message):
  - Als Gegensatz zu bestätigenden Nachrichten gibt es auch Nachrichten die nicht bestätigt werden müssen.
  - Dies trifft auf Nachrichten zu, die für bestimmte Anwendungsanforderungen häufiger wiederholt werden müssen, wie zum Beispiel wiederholtes Lesen eines Sensors.
- Bestätigungsnachricht (Acknowledgement Message):
  - Eine solche Nachricht bestätigt den Empfang einer bestätigenden Nachricht. Eine Bestätigungsnachricht sagt nicht aus, ob die Anfrage, die mit einer bestätigenden Nachricht versendet wurde, erfolgreich war oder nicht.
  - Jedoch enthält die Bestätigungsnachricht auch eine sogenannte *Piggybacked Response*.
- Rücksetzende Nachricht (Reset message):
  - Diese Nachricht sagt aus, dass eine spezifische Nachricht (*Confirmable* oder *Non-confirmable*) empfangen wurde, jedoch einige Teile des Nachrichtenkontextes fehlt, um die richtig zu bearbeiten.

  - Dieses Verhalten tritt auf, wenn der zu empfangende Endpunkt neu gestartet und somit den Zustand vergessen hat, der zur vollständigen Interpretation der Nachricht nötig ist.
  - Ein absichtliches Provozieren einer rücksetzenden Nachricht *Reset Message*, zum Beispiel durch das Senden einer leeren bestätigenden Nachricht (*Empty Confirmable Message*), kann als eine kostengünstige Prüfung der Funktionsfähigkeit eines Endpunktes verwendet werden - vergleichbar mit einem *Ping*.

- Piggybacked Response:
  - Eine *Piggybacked Response* ist direkt in eine *Acknowledgement* (ACK) Nachricht inkludiert, die gesendet wird, um den Empfang der Anfrage für diese Antwort zu bestätigen.

- Separate Antwort (Separate Response):
  - Wenn eine *Confirmable* Nachricht mit einer Anfrage mit einer *Empty* Nachricht quittiert wird (z.B. weil der Server die Antwort nicht sofort hat), wird eine separate Antwort in einem separaten Nachrichtenaustausch gesendet.

- Leere Nachricht (Empty Message):
  - Eine Nachricht mit dem Code 0.00. Die Nachricht ist weder eine Anfrage noch eine Antwort. Es beinhaltet nur den 4-Byte-langen Kopf (*header*).

Nachrichtenübertragung

CoAP nutzt zur Nachrichtenübertragung UDP, um den Austausch von Nachrichten asynchron ausführen zu können. Dies wird dadurch erreicht, dass eine weitere Schicht auf UDP aufbauend eingefügt wird (siehe Bild 1). Diese Schicht kann optional auch einen Mechanismus zur Sicherstellung des Nachrichtenaustausches beinhalten. Dabei definiert CoAP vier verschiedene Arten von Nachrichten:

- bestätigende Nachricht (*Confirmable Message*)
- nicht bestätigende Nachricht (*Non-confirmable Message*)
- Bestätigungsnachricht (*Acknowledgement Message*)
- rücksetzende Nachricht (*Reset Message*)

Diese vier Arten stehen orthogonal zueinander, sprich:

- Bestätigende und nicht bestätigende Nachrichten sind für Anfragen (*requests*)
- und rücksetzende Nachrichten oder Nachrichten, die eine Anfrage bestätigen, sind für Antworten (*responses*) gedacht.

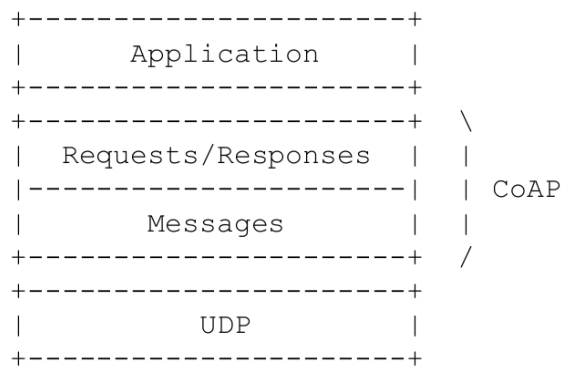

Abbildung 1: Abstrakte Darstellung der verschiedenen Schichten

Das Nachrichtenmodell des Constrained Application Protocols basiert auf den Austausch von Nachrichten über UDP. Dabei beginnt die Nachricht mit einem in der Länge fixierten Vier-Byte-Langen Kopfzeile (*header*), gefolgt von einem optionalen Token (null bis acht Bytes lang), null oder mehr sogenannten Optionen. Diese Optionen sind vergleichbar mit den *header fields* von HTTP. Nach den Optionen befindet sich ein sogenannter Anhang-Markierer (*Payload marker*) der einem Byte mit dem Wert 0xFF (255) entspricht, jedoch nur in der Nachricht enthalten ist, wenn eine Payload mitgesendet wird. Anschließend an den *Payload marker* kommt der Anhang (*Payload*). Dieses Format ist sowohl für Anfragen als auch für Antworten dasselbe.

Damit Nachrichten als eindeutig identifiziert werden können, wird ein sogenannter Nachrichtenbezeichner (*Message ID*) verwendet. Dieser ist 16 Bit groß und erlaubt somit, bei Implementierungen mit Standardeinstellungen, bis zu 250 Nachrichten die Sekunde von einem Endpunkt zu einem anderen zu senden. Die *Message ID* wird auch für die Sicherstellung des Nachrichtenaustausches (*reliability*) benötigt. Dabei ist jedoch die *Message ID* nur zwischen zwei Endpunkten eindeutig. Kommuniziert ein Teilnehmer mit mehreren Teilnehmern gleichzeitig, dann können die *Message IDs* häufiger vorkommen. Für die eindeutige Identifizierung der Kommunikation zwischen zwei Teilnehmern wird der sogenannten Token verwendet. Dieser ist über mehrere Verbindungen hinweg eindeutig und kann als Identifikator für Verbindungen gesehen werden.

Die Sicherstellung des Nachrichtenaustausches erfolgt dadurch, dass man eine Nachricht als bestätigend (*Confirmable*) markiert. Eine als *Confirmable* gekennzeichnete Nachricht wird so lange an den jeweiligen Empfänger gesendet, bis dieser eine *Acknowledgement* Nachricht mit derselben *Message ID* zurücksendet (wie in Bild 2 dargestellt). Wenn der Empfänger die *Confirmable* Nachricht, aufgrund fehlender Daten oder fehlendem Kontext, nicht beantworten kann, sendet dieser eine *Reset* Nachricht zurück.

Jedoch wird für den Austausch von Nachrichten im CoAP Kontext kein Sicherheitsmechanismus für die Übertragung gefordert, sondern es können auch Nachrichten als *Non-confirmable* markiert werden (siehe Bild 3). Dies bietet sich zum Beispiel an, wenn

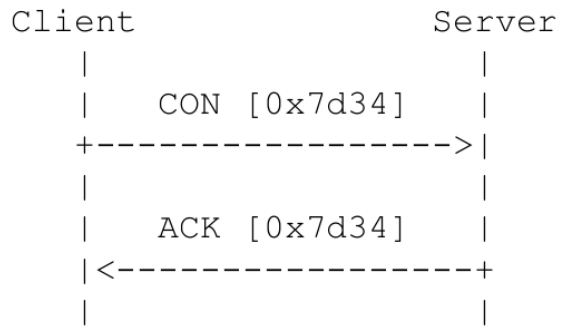

Abbildung 2: Nachrichtenaustausch mit Sicherstellung des Transfers (Quelle: [3]).

man die Messdaten eines Sensors wiederholt ausliest. Dabei werden *Non-confirmable* Nachrichten nicht bestätigt, jedoch wird eine *Message ID* benutzt, um Duplikate zu erkennen.

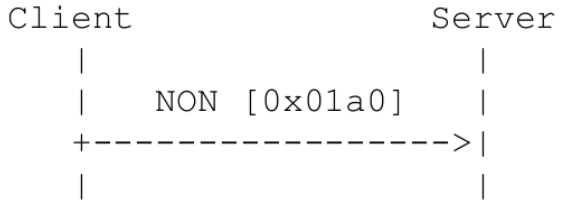

Abbildung 3: Nachrichtenaustausch ohne Sicherstellung des Transfers (Quelle: [3]).

Kann eine einkommende Anfrage (*Request*), die mithilfe einer *Confirmable* Nachricht versendet wurde, sofort beantwortet werden, wird die Antwort (*Response*) in der daraus resultierenden *Acknowledgment* Nachricht zurückgesendet. Dieses Prinzip nennt man auch *Piggybacked Response*. Das Bild 4 stellt diesen Mechanismus dar.

Ist der Server jedoch nicht sofort in der Lage die Anfrage zu beantworten, dann antwortet dieser mit einer leeren *Confirmable* Nachricht. Dies tut er, um den Client vom wiederholten Senden der Anfrage zu stoppen. Sind alle benötigten Daten zur Beantwortung der Anfrage vorhanden, sendet der Server die Antwort in einer neuen *Confirmable* Nachricht. Dieses Prinzip wird als separate Antwort (*separate response*) bezeichnet und kann mit dem Bild 5 nachvollzogen werden.

Dabei macht CoAP Gebrauch von den bekannten Internetmethoden *GET*, *PUT*, *POST* und *DELETE* in einer ähnlichen Art wie es HTTP vollzieht.

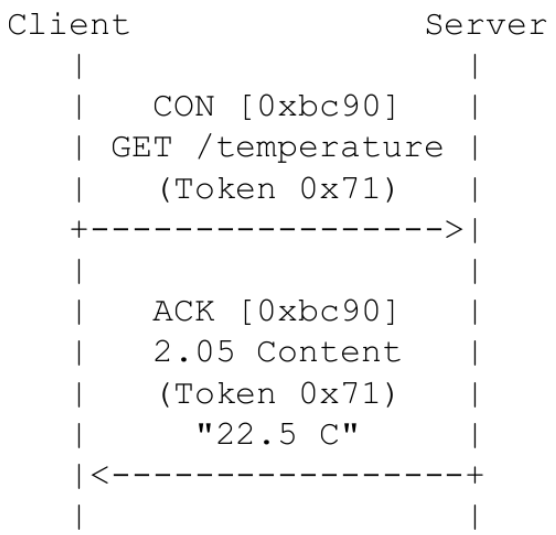

Abbildung 4: Beispiel eines erfolgreichen *Piggybacked Response* (Quelle: RFC 7252 von Shelby, Hartke und Bormann [3]).

Nachrichtenformat

Wie schon erwähnt, basiert der Nachrichtenaustausch von CoAP auf UDP. Dabei nimmt jede, über UDP versendete Nachricht ein ganzes UDP Datagramm in Anspruch. Dabei ist der Aufbau einer CoAP Nachricht einfach gehalten und startet mit einer Vier-Byte-langen-Kopfzeile (*Header*). Diese beinhaltet folgende Daten (visualisiert im Bild 6):

- *Version*
- *Type*
- *Token Length*
- *Code*
- *Message ID*

Dabei repräsentieren die ersten zwei Bits des *Headers* die Versionsnummer. Die Versionsnummer gibt an, welcher CoAP Version die Nachricht erstellt wurde bzw. verarbeitet werden kann. In dieser Arbeit beschäftigen wir uns mit CoAP Nachrichten mit der Versionsnummer 1 (01 in Binär).

Die darauffolgenden zwei Bits entsprechen dem Typ der CoAP Nachricht. Der Typ gibt an, ob es sich um eine *Confirmable* (0 = 00 in Binär), *Non-Confirmable* (1 = 01), *Acknowledgement* (2 = 10) oder *Reset* (3 = 11) Nachricht handelt.

Als Nächstes kommt die vier Bit lange *Token Length*, die die Länge des *Tokens* angibt. Dabei kann der *Token* zwischen null (0000 in Binär) und acht (0111) Bytes lang sein.

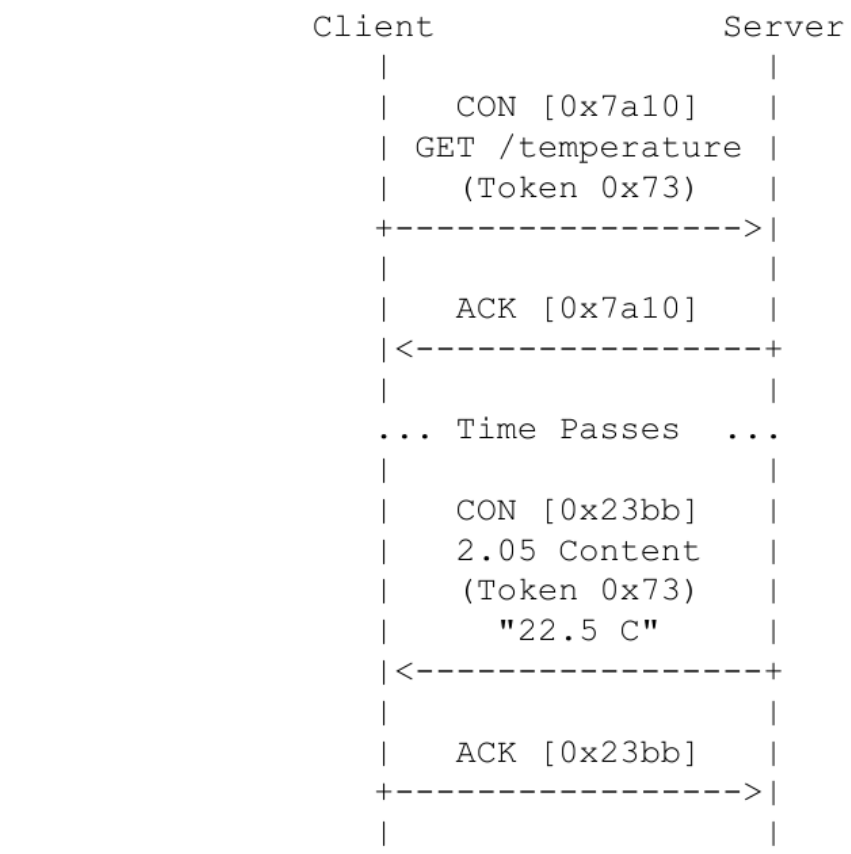

Abbildung 5: Beispiel einer *separate response* (Quelle: RFC 7252 von Shelby, Hartke und Bormann [3]).

Token Lengths zwischen neun und fünfzehn Bytes sind im RFC 7252 [3] für zukünftige Versionen reserviert.

Die nächsten 8 Bit geben den *Code* (vgl. mit dem Statuscode bei HTTP) der CoAP Nachricht an. Dabei unterteilt sich der *Code* in eine drei Bit lange *Code Class* (*most significant bits*) und einen fünf Bit lange *Code Detail* (*least significatn bits*). Dabei folgt der *Code* dem Schema "c.dd", wobei "c"Werte von 0 bis 7 annehmen kann und "dd"Werte von 00 bis 31. Die *Code Class* gibt dabei an, ob es sich um

- eine Anfrage (0),
- eine erfolgreiche Antwort (2),
- eine clientseitige, fehlerhafte Antwort (4),
- oder eine serverseitige, fehlerhafte Antwort (5).

Dabei nimmt der *Code* 0.00 eine besondere Stellung ein, da dieser eine leere Nachricht (*Empty Message*) markiert. Die *Codes* gleichen sich mit einigen Statuscodes, die man von HTTP kennt, jedoch ist nicht jeder Statuscode als CoAP *Code* abgebildet.

Der letzte Teil des *Headers* ist die sogenannte *Message Id*, die 16 Bit in Anspruch nimmt und in der *Networt Byte Order* (*Big Endian*) angegeben wird. Ihre Aufgabe ist es, Duplikate von Nachrichten zu erkennen. Auch wird die *Message Id* dazu benutzt, um Nachrichten vom Typ *Acknowledgement* und *Reset* zu Nachrichten vom Typ *Confirmable* und *Non-Confirmable* zu verlinken. Somit besitzt der Server immer einen Überblick, zu welcher Anfrage schon eine Antwort geschickt wurde.

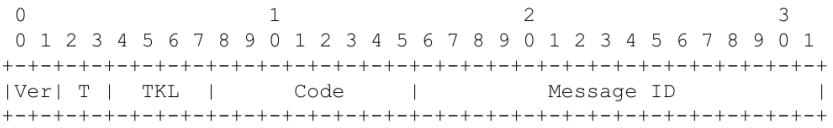

Abbildung 6: Binäre Struktur eines CoAP *Headers* (Quelle: [3]).

Anschließend an den *Header* kommt der *Token* der Nachricht. Dieser ist in seiner Länge variabel und hängt von der im *Header* angegebenen *Token Length* ab. Dieser ist zuständig für die Korrelation von Anfragen zu Antworten.

Nachfolgend können null oder mehr sogenannte *Options* folgen. Der Option können folgende Bestandteile einer CoAP Nachricht folgen:

- das Ende der CoAP Nachricht (EoF),
- eine weitere Option,
- oder der *Payload Marker* mit anschließender *Payload*.

Ist eine *Payload* gegeben, folgt nach der Gruppe von *Options* ein sogenannter *Payload Marker*. Dieser besteht aus einem Byte voller logischer Einsen (0xFF) und markiert somit das Ende der *Options*. Alle Daten, die nach dem *Payload Marker* befinden, werden als *Payload* behandelt. Dabei ist die Länge durch die *UDP Datagram* Paketgröße von 65535 Bytes begrenzt. Wird für die Übertragung der Nachricht mehr Bytes benötigt, als ein *UDP Datagram* an Größe bereitstellen kann, werden die Bytes auf mehrere *UDP Datagrams* aufgespalten. Diesen Mechanismus nennt man auch *Blockwise transfer* und wird im RFC 7959 von Shelby und Bormann [2] beschrieben.

Aufbau einer Option

Eine Option wird durch eine eindeutige Nummer identifiziert. Neben der Nummer besitzt eine Option auch einen Wert (*Value*), den diese Option hält, und einen Indikator für die Länge des Wertes. Dabei wird die Nummer nicht direkt in die Nachricht kodiert, sondern die Options werden zuerst aufsteigend nach ihrer Nummer sortiert und dann wird eine Deltakodierung (Differenzbildung) zwischen der aktuellen Option und deren Vorgängern

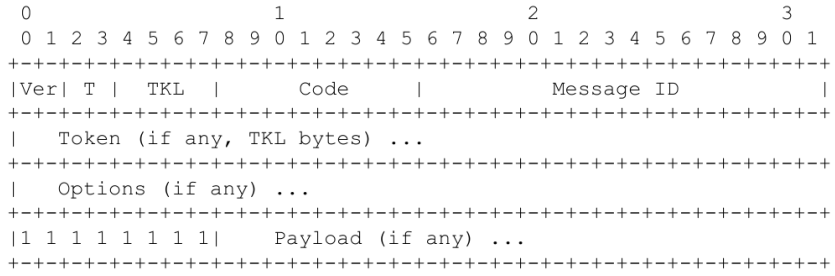

Abbildung 7: Binäre Struktur einer vollständigen CoAP Nachricht (Quelle: [3]).

gebildet. Dies geschieht dadurch, dass alle vorherigen Differenzen (*Deltas*) addiert werden und dann die Differenz zur aktuellen Option gebildet wird. Für die erste Option wird der Sonderfall behandelt, dass als vorheriges Delta ein Wert von null angenommen wird. Dies resultiert darin, dass für die erste Option die kodierte Differenz als Nummer der Option verwendet wird. Ein weiterer Sonderfall ist derjenige, wenn mehrere Instanzen der gleichen Option in der Kollektion von Options auftritt. Dabei ist die Differenz zwischen zwei gleichen Options immer null.

Eine Option fängt immer mit einem Byte an, das zwei Informationen enthält. Einmal die Differenz (*Option Delta*) und die Länge des Wertes (*Option Length*) der Option. Das *Option Delta* entspricht dabei den ersten vier Bits (*most significant bits*) und die *Option Length* die letzten vier Bits (*least significant bits*). Man kann dieses Byte auch als einen *Header* für Options bezeichnen, da dieser Informationen beinhaltet, die für das Kodierung bzw. Dekodieren von Options benötigt wird.

Um jedoch Differenzen und Längen, jenseits des Wertes fünfzehn, verwenden zu können, können dem *Option Header* das *Option Delta (extended)* und *Option Length (extended)* folgen. Diese beiden können jeweils zwischen null und zwei Bytes lang sein. Nach diesen beiden folgt der *Option Value*, der null oder mehr Bytes betragen kann.

1.2 Ausführungsparadigmen in der Informatik

In der Informatik spricht man von zwei großen Ausführungsparadigmen, mit denen Programme bzw. Codeteile ausgeführt werden können: **synchrone** und **asynchrone Ausführung**. Diese unterscheiden sich in wesentlichen Punkten deutlich voneinander und bieten in verschiedenen Einsatzszenarien Vor- und Nachteile (siehe Tabelle 1). Dabei unterstützen die geläufigsten Programmiersprachen von Haus aus eine synchrone Ausführung von Programmen, jedoch bieten nicht alle eine asynchrone Ausführung.

Um diese Unterschiede zwischen den beiden Paradigmen zu veranschaulichen, wird dies anhand einer Client-Server-Anwendung mit einer an den Server angeschlossenen Datenbank und zwei Clients verbildlicht (siehe Bild 8 und 9). Dabei ist der Server eine einfache Web-Applikation, die Daten mittels SQL von der angeschlossenen Datenbank holt.

	Synchron	Asynchron
Programmfluss	Stoppt den Programmfluss.	Kann im Programmfluss weiter gehen.
Beendigung	Überprüft periodisch, ob Funktion beendet ist.	Ein Event markiert die Beendigung der Funktion.
Main-Thread	Ist als ”Blockedöder ”Waiting”markiert.	Ist frei für andere Aufgaben.

Tabelle 1: Vergleich zwischen synchroner und asynchroner Ausführung

Senden nun beide Clients, in kurzem Zeitabstand zueinander, eine Anfrage an den Server, dann kann der synchrone Server nur die Anfrage bearbeiten, die zuerst eintrifft - in diesem Fall die des Client 1. Dies geschieht deswegen, da der Main-Thread bzw. der für das Empfangen der Pakete zuständige Thread auf die Rückantwort des Datenbankservers wartet. Dadurch wird die Anfrage von Client 2 auf dem Server zurückgehalten und erst bearbeitet, wenn die erste Anfrage bearbeitet und zurückgesendet wurde. Dieses Verhalten skaliert schlecht mit mehreren, gleichzeitig eintreffenden Anfragen, da der sogenannte *Threadpool*[1], bei zu vielen, langandauernden Anfragen, seine volle Kapazität erreicht und somit keine neuen Anfragen / Aufgaben annehmen wird.

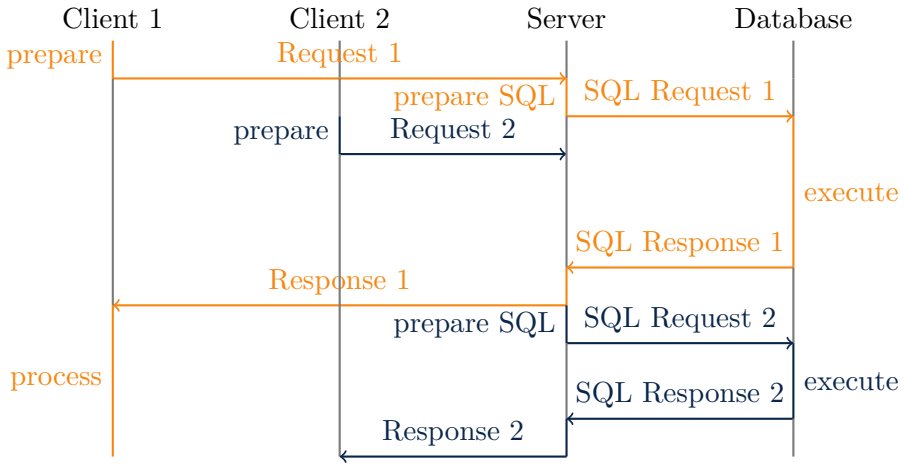

Abbildung 8: Sequenzdiagramm eines synchronen Servers

Wird nun statt einem synchronen Server ein asynchroner Server eingesetzt, ändert sich das beschriebene Szenario folgendermaßen: Trifft die Anfrage des Client 1 ein, wird diese, wie zuvor, sofort abgearbeitet. Jedoch geschieht dies auf einen anderen Thread, damit der Main-Thread bzw. der für das Empfangen der Pakete zuständige Thread wieder frei ist. Wird nun vom Client 2 eine Anfrage verschickt, dann kann der Server diese entgegennehmen und auf einen weiteren, freien Thread verlagern und bearbeiten. Je nachdem welche Anfrage schneller vom Datenbankserver bearbeitet wird, in diesem Fall die des Client 1, wird der Main-Thread bzw. der Thread der das Paket zuvor entgegengenommen hat, über die Beendigung der Datenbankanfrage benachrichtigt. Somit kann dieser

[1] Entspricht einer Queue in der die zu bearbeitenden Aufgaben abgelegt werden.

Thread seinen Kontext synchronisieren und die Antwort an den Client 1 zurücksenden.

Dieses Verfahren skaliert deutlich besser mit steigender Anzahl von Anfragen, jedoch kann dies auch mehr CPU- und Speicherressourcen verbrauchen. Dies hat den Grund, da zum Aufrechterhalten des Zustandes eine asynchrone Zustandsmaschine konstruiert wird und der Kontextwechsel bei Beendigung des asynchronen Aufrufs mit den Main-Thread synchronisiert werden muss.

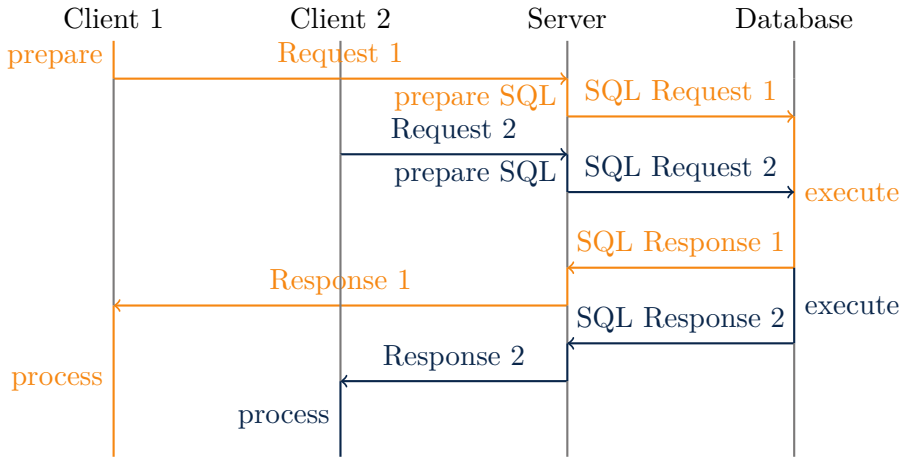

Abbildung 9: Sequenzdiagramm eines asynchronen Servers

Asynchronität in C#

Mit *Task-based Asynchronous Pattern* (TAP) ermöglicht Microsoft in C# Asynchronität. Dieses Pattern ermöglicht eine einfache Transformation von synchronen Code zu asynchronen Code ohne große Änderungen vornehmen zu müssen. Auch ist es von Haus aus im Sprachkonstrukt von C# integriert und kann somit ohne zusätzliche Konfiguration verwendet werden.

Dabei baut das asynchrone Ausführungsparadigma für C# auf folgende Komponenten auf:

- `Task`: Ermöglicht eine asynchrone Methode zu definieren.
- `Task<TResult>`: Ermöglicht einen Rückgabewert vom Typ TResult von einer asynchronen Methode zurückzugeben.
- `CancellationToken`: Ermöglicht es einen Aufruf einer asynchronen Methode vorzeitig zu beenden.
- `async`/`await`: Schlüsselwörter um asynchrone Methoden zu deklarieren und auszuführen.

In dem nachfolgenden Codeausschnitt 1 wird eine Instanz eines DownloadClient erzeugt. Dieser hat eine Methode, mit der der Inhalt einer Webseite heruntergeladen und als `string` zurückgegeben werden kann.

```
1 public string Download(Uri uri) {
2     var client = new DownloadClient();
3     var result = client.Download(uri);
4
5     return result;
6 }
```

Listing 1: Synchrone Methode in C#

Dies hat den Nachteil, dass die Funktion den Main-Thread bzw. den Thread, der diese Methode aufgerufen hat, solange blockiert bis der gesamte Webseiteninhalt der angegebenen URI heruntergeladen wurde. Bei UI-Anwendungen lässt sich dadurch die Oberfläche nicht mehr bedienen und friert ein. Bei Verwendung in einer Konsolen-Applikation ohne UI ist der Main-Thread bis auf Weiteres blockiert und somit können andere Programmteile, die nicht auf das Ergebnis dieser Methode warten, nicht ausgeführt werden.

Damit der aufzurufende Thread bzw. der Main-Thread nicht andauernd auf die Beendigung der Methode warten muss, kann man Events einsetzen (siehe Codebeispiel 2). Diese beendigen vorzeitig die Exekution der Methode, indem der Aufrufer ein Objekt übergeben bekommt. Dieses Objekt repräsentiert den derzeitigen Zustand der Operation - also in diesem Fall das Herunterladen des Webseiteninhaltes. Dafür wird in Zeile 3 das `DownloadResult`-Objekt erzeugt. In Zeile 5 wird durch den Operator `+=` die nachfolgende anonyme Lambdafunktion `(content) => result.SetComplete(content)` als Beobachter des Events `client.DownloadComplete` registriert. Diese Lambdafunktion hat die Aufgabe den heruntergeladen Inhalt der Webseite in das DownloadResult zu setzen und als vollständig zu markieren. Dies passiert durch den Aufruf der Methode `SetComplete(string content)` auf dem DownloadResult-Objekt. Anschließend startet der DownloadClient die Operation mithilfe des Aufrufs `client.StartDownload(uri)`.

Ist der Client mit dem Download fertig, dann feuert dieser das Event `DownloadComplete` und benachrichtigt somit alle darauf hörenden Empfänger - in diesem Fall die anonyme Lambdafunktion.

Der aufzurufende Thread hat nun die Möglichkeit andere Funktionen auszuführen, solange der Download noch nicht beendet ist.

```
1 public DownloadResult Download(Uri uri) {
2     var client = new DownloadClient();
3     var result = new DownloadResult();
4 
5     client.DownloadComplete += (content) => result.SetComplete(content);
6     client.StartDownload(uri);
7 
8     return result;
9 }
```

Listing 2: Eventbasierte Methode in C#

Als nächste Ausbaustufe der Funktion gibt es noch die asynchrone Variante. Hierbei wird der DownloadClient um eine asynchrone Downloadmethode, namentlich DownloadAsync, erweitert. Somit kann das Herunterladen des Webseiteninhaltes völlig asynchron und auf einem dezidiertem Thread passieren, ohne den Main-Thread bzw. aufzurufenden Thread zu beeinträchtigen. Dies kann jedoch nur solange ausgenutzt werden, solange keine andere Methode auf das Ergebnis der Downloadoperation angewiesen ist. Das Codebeispiel veranschaulicht diese asynchrone Methode und verbildlicht auch die geringen Änderungen zur synchronen Implementierung diese Methode (siehe Codebeispiel 3). Dieses Pattern ist auch als *Task-based asynchronous pattern*, abgekürzt TAP, bekannt.

```
1 public async Task<string> DownloadAsync(Uri uri, CancellationToken ct) {
2     var client = new DownloadClient();
3     var result = await client.DownloadAsync(uri, ct).ConfigureAwait(false);
4 
5     return result;
6 }
```

Listing 3: Asynchrone Methode in C#

Hierbei ist auch zu erwähnen, dass eine Methode sich immer durch die Schlüsselwörter `async`/`await` auszeichnet. Dabei wird nur `async` vorausgesetzt, wenn innerhalb der asynchronen Methode auf ein Ergebnis einer anderen asynchronen Methode, mittels dem Schlüsselwort `await`, gewartet wird (siehe Zeile 3).

1.3 Bekannte Implementierungen

Für fast jede Sprache gibt es eine Implementierung von CoAP. Die bekannteste von allen ist hierbei Californium für Java[2]. Andere Implementierungen gibt es, auszugsweise, für folgende Sprachen:

- C#: CoAP.NET
- Python: CoAPython
- C: libcoap
- Javascript: Copper
- Go: Go-Coap

Californium

Die bekannteste Implementierung ist hierbei Californium für Java. Es wird von der Eclipse Foundation gefördert und von mehr als 50 Entwicklern gepflegt. Aufgrund der aktiven Entwicklung und großen Abdeckung der jeweiligen RFC Standards für CoAP wurde Californium für verschiedene Sprachen portiert - z.B. mit CoAP.NET für .NET und C#.

CoAP.NET

CoAP.NET implementiert das Constrained Application Protocol in C# und wurde von der Organisation *smeshlink* bis Juli 2016 entwickelt. Es ist ein C#-Port der CoAP-Implementation für Java *Californium*, jedoch ist es zurzeit unbetreut. Auch eine vollständige Unterstützung von asynchronen Techniken ist nicht in CoAP.NET gegeben. Dennoch zählt es zu den bekanntesten Implementationen im C#-Umfeld und wird auch in der Softwareentwicklungsfirma *World-Direct eBusiness solutions GmbH* verwendet. Da diese Bachelorarbeit in Kooperation mit dieser Firma entstanden ist, wurde ein großer Fokus auf CoAP.NET gelegt.

1.4 Ziel der Arbeit

Die Forschungsfrage, die diese Bachelorarbeit zu beantworten versucht, lautet: *Hat eine asynchrone Implementation eines Servers Einfluss auf dessen Durchsatzrate?*

Um diese Frage ursprünglich zu beantworten, versuchte man in CoAP.NET eine komplette asynchrone Unterstützung zu implementieren und dann die synchrone Schnittstelle

[2] https://www.eclipse.org/californium/

gegen die asynchrone Schnittstelle zu vergleichen. Dies hat sich als nicht rentabel herausgestellt, da zu viele Änderungen an der Codebasis von CoAP.NET vorzunehmen waren und somit auch der Vergleich nicht verhältnismäßig gewesen wäre.

Darum fiel die Wahl auf eine komplette Neuentwicklung des Constrained Application Protocols in C# mit modernen Technologien und Patterns. Dabei wurde bei der Entwicklung auf eine synchrone als auch asynchrone Schnittstelle Wert gelegt, um sowohl synchron als auch asynchron zu vergleichen. Somit sollte der Einfluss bewertet und aufgezeigt werden, ob eine asynchrone Implementation eines Servers einen merklichen Einfluss auf dessen Durchsatzrate besitzt.

Das Resultat dieser Arbeit ist im Github Repository world-direct/CoAP.NET einsehbar.

2 Implementierung

Das Constrained Application Protocol wird für diese Bachelorarbeit in der Programmiersprache C# implementiert. Dabei wird die Implementierung nach dem .NET Standard 2.1 entwickelt. Diese ermöglicht es die Bibliothek lauffähig unter den folgenden .NET Versionen und Laufzeitumgebungen zu verwenden: .NET 5.x und .NET Core 3.x. Somit sind alle aktuellen und zukünftig unterstützten Laufzeitumgebungen abgedeckt. Durch .NET Core ist auch eine Verwendung abseits des Betriebssystems Windows möglich.

Die Asynchronität ist, wie schon beschrieben, durch das Sprachkonstrukt gegeben. Somit kann mittels TAP mit wenig Aufwand eine asynchrone API bereitgestellt werden. Zusätzlich wird die Drittanbieterbibliothek *Task Parallel Library*, auch TPL abgekürzt, genutzt. Diese Bibliothek erlaubt es ein sogenanntes Dataflow-Mesh (vergleichbar mit einer Pipeline, nur mit erweiterter Funktionalität und mit Asynchronität als Hauptfokus) aufzubauen und somit eine asynchrone Datenverarbeitung innerhalb einer Applikation zu ermöglichen. Dabei besteht ein solches Dataflow-Mesh aus sogenannten *Dataflow Blocks*. Die vordefinierten Dataflow Blocks geben entweder die Möglichkeit Daten zu puffern (*Buffering Blocks*) oder zu verarbeiten (*Execution Blocks*). Als weitere Möglichkeit bietet TPL an eigene Dataflow Blocks zu definieren, indem man die entsprechenden Basisklassen oder Interfaces implementiert.

Gefundene Fehler

Nachfolgend sind hier Bugs aufgeführt, die innerhalb dieser Bachelorarbeit in CoAP.NET gefunden wurden:

- Blockweiser Transfer propagiert keine Fehler. Das heißt, dass eine Übertragung von mehreren UDP-Paketen nicht gestoppt wird, wenn ein Statuscode von 4.08 (Request Entity Incomplete) oder 4.13 (Request Entity Too Large) zurückgegeben wird.
- Der Client stoppt die Neuversendung von Anfragen nicht, obwohl eine Antwort mit der passenden MessageId zurückgesendet wurde.

2.1 Struktur der Applikation

Die Applikation gliedert sich in folgende Komponenten:

1. Transports: Eine Transport-Klasse übernimmt das Senden und Empfangen von CoAP-Nachrichten auf dem jeweiligen Protokoll. Zum Beispiel ist die *UdpTransport*-Klasse verantwortlich für das Senden und Empfangen von CoAP-Nachrichten, die mittels UDP übertragen werden.

2. Channels: Ein Channel repräsentiert eine aktive Verbindung zwischen einem CoAP-Client und CoAP-Server über ein beliebiges Protokoll. Für UDP gibt es z.B. eine *UDPChannel*-Klasse, die auf einem vordefinierten Port auf UDP-Pakete horcht und über diesen Antworten an den jeweiligen Client zurücksendet.
3. Serializers: Ein *Serializer* bietet Methoden für das De- als auch Serialisieren von CoAP-Nachrichten, für ein bestimmtes Nachrichtenformat und/oder eine CoAP-Version an. Dabei erhalten diese die Daten entweder von einem der *Channels* oder von einer Ressource, die auf dem Server registriert ist.
4. Handlers: Ein Handler kümmert sich um die Verarbeitung und Weiterleitung von CoAP-Request an die jeweiligen Ressourcen oder von CoAP-Responses an den *Serializer*.
5. Ressourcen: Eine Ressource ist vergleichbar zu einem HTTP- bzw. Controller-Endpunkt. Diese registriert sich beim Server unter einer definierten URI und gibt an, welche Methoden (GET, POST, PUT, DELETE) diese anbietet.

Dabei stellt jede Komponente einen *Dataflow Block* dar. Somit kann eine asynchrone Weiterleitung zwischen den einzelnen Komponenten sichergestellt werden. Dies geschieht dadurch, dass die einzelnen Blöcke miteinander verknüpft und auf ankommende Daten in einer asynchronen Weise gewartet werden. Damit können Daten, sobald diese verfügbar sind, umgehend verarbeitet werden. Auch übernimmt das Dataflow-Mesh die Verantwortung für die angewendete Parallelität und Synchronität der Daten. Somit kann eine flexible und anpassbare Verarbeitungskette implementiert werden, die vollkommen Asynchron arbeitet.

Die API des Serializers ist inspiriert von der API des Sytem.Text.Json Serializers für JSON Dateien, der von Microsoft entwickelt wird. Der CoapMessageSerializer bietet dabei folgende Schnittstellen an:

- `void CoapMessage Deserialize(ReadonlySpan<byte> value);`
- `Task<CoapMessage> DeserializeAsync(Stream value, CancellationToken ct);`
- `void byte[] Serialize(CoapMessage message);`
- `Task SerializeAsync(Stream stream, CoapMessage message, CancellationToken ct);`

3 Messung

Zur Messung der Performance und Durchsatzrate der Implementierung betrachten wir nur den Serializer, da auf der Empfangsseite auf die bereits bestehenden .NET-Implementierungen der Sockets zur Datenübertragung von den UDP- bzw. TCP-Paketen gesetzt wird. Dabei wird auf TCP als Übertragungsweg gesetzt, da sich bei UDP die Paketgröße nicht verändern lässt bzw. auf maximal 2^{16} Bits (= 65.535 Bytes) beschränkt ist. Somit können wir die Paketgröße der CoAP-Nachricht, aufgrund der fehlenden Implementierung des blockweisen Transfers, nicht beliebig erhöhen. Um hier jedoch ein Szenario zu kreieren, indem wir auch sehr lange Nachrichten übermitteln können, wurde die Übertragung von CoAP-Nachrichten über TCP nach dem RFC 8323 von Bormann u. a. [1] implementiert.

3.1 Nachrichtenverarbeitung

In diesem Szenario wird die Zeit der Nachrichtenverarbeitung auf dem Server gemessen. Dabei sieht die Messung folgenden Ablauf vor:

- Die Nachrichten werden von einem Client über das jeweilige Protokoll über TCP an den CoAP-Server versendet.
- Dabei wird eine langsame Übertragung simuliert, indem nur ein Byte alle 250 Millisekunden (4 B/s) verschickt wird.
- Der Server hört für TCP in der asynchronen Variante auf den Port 5683 und für die synchrone Variante auf Port 5684.
- Dabei wird die Zeit gemessen, wie lange das Verarbeiten der Nachricht gedauert hat.
- Die Zeitmessung wird am Client gestartet, sobald der Client mit dem Versenden der Nachricht beginnt. Gestoppt wird diese, sobald der Server die Nachricht deserialisiert hat.
- Dieser Ablauf wird mehrere Male wiederholt und dann der Durchschnittswert ermittelt.

BenchmarkDotnet

BenchmarkDotnet ist ein Software-Tool das vom dotnet-Team[3] entwickelt und zur Verfügung gestellt wird. Mit diesem Tool lassen sich automatisierte Laufzeittests von einem bestimmten Codeteil oder sogar einem ganzen Programm erzeugen.

[3] Open-Source-Abteilung bei Microsoft für das .NET Ökosystem

BenchmarkDotnet führt dafür den ausführenden Codeteil in mehreren Durchläufen aus und misst bei jedem Durchlauf verschiedene Parameter, die vom Nutzer festgelegt werden. Dabei wird standardmäßig die durchschnittliche Laufzeit, die Fehlertoleranz und Standardabweichung ermittelt. Auch können Parameter wie allokierten Speicher, Codeverlauf (Tracer), Anzahl von Lock's (Semaphore), Anzahl verarbeiteter Aufträge im Threadpool und vieles mehr aufgezeichnet werden.

Die Ergebnisse werden dabei in verschiedene Formate exportiert. Standardmäßig werden diese in CSV, HTML und Markdown exportiert, jedoch stehen auch JSON, XML und auch als grafische Visualisierung in RPlot.

3.2 Serialisierung und Deserialisierung

In diesem Szenario wird die Verarbeitungszeit der synchronen als auch asynchronen Variante der Serializierungs- bzw. Deserialisierungsmethode gemessen. Dies wird mittels der Bibliothek BenchmarkDotnet[4] durchgeführt. BenchmarkDotnet ist dabei ein Tool, dass es erlaubt nativ in C# eine vordefinierte Methode bzw. einen bestimmten Teil eines Programms einem Benchmark zu unterziehen. Somit wird ermittelt, wie sich diese beiden Varianten, unabhängig von Netzwerkgeschwindigkeit, verhalten. Dabei wird sowohl die ermittelte Durchschnittszeit als auch der Speicherverbrauch mittels BenchmarkDotnet gemessen.

Dabei können wir in BenchmarkDotnet mehrere verschiedene Durchläufe konstruieren, die sich in bestimmten Parametern unterscheiden. Um dabei die Länge der CoAP-Nachricht zu variieren, wird die Anzahl der Options und die Länge der Payload verändert. Dabei verändern sich die beiden Parameter in folgenden Schritten: 0, 1000, 100000. Somit sollten folgende Fälle abgedeckt sein:

- Eine CoAP-Nachricht nur mit dem Header und dem Token.
- Eine CoAP-Nachricht nur mit Options und keiner Payload.
- Eine CoAP-Nachricht nur mit einer Payload und keinen Options.
- Eine CoAP-Nachricht sowohl mit Options als auch einer Payload.

Auch sollte ersichtlich sein, unter welchen Umständen synchron oder asynchron besser abschneidet.

Dabei entspricht ein Benchmark eine auszuführende Methode bzw. ein auszuführender Codeteil. Ein Benchmark wird dabei folgendermaßen deklariert:

[4] https://benchmarkdotnet.org/index.html

```
public class BenchmarkExample
{
    // Initialisierung eines Zufallgenerators.
    private static readonly Random Random = new Random();
    private readonly byte[] data;
    private readonly SHA256 sha256 = SHA256.Create();
    private readonly MD5 md5 = MD5.Create();

    // Initialisierung der Daten für den Benchmark
    public BenchmarkExample()
    {
        this.data = new byte[10000];
        this.Random.NextBytes(this.data);
    }

    // Deklarierung eines Benchmarks für SHA256 Berechnung.
    [Benchmark]
    public byte[] SHA256 => this.sha256.ComputeHash(this.data);

    // Deklarierung eines Benchmarks für MD5 Berechnung.
    [Benchmark]
    public byte[] MD5 => this.md5.ComputeHash(this.data);
}
```

Listing 4: Beispiel eines Benchmarks in BenchmarkDotnet

In der Main-Methode muss diese Klasse nun nur bei BenchmarkDotnet zur Ausführung registriert werden. Dies geschieht folgendermaßen:

```
public class Program
{
    public static void Main(string[] args)
    {
        var summary = BenchmarkRunner.Run<BenchmarkExample>();
    }
}
```

Listing 5: Ausführen der Benchmark-Klasse

Nachrichtengenerierung für Benchmark

Die CoAP-Nachrichten, die für den Benchmark benutzt werden, folgen folgendem Schema (x = Anzahl der Options; y = Länge der Payload in Bytes):

- Die CoAP-Version ist immer auf 1.
- Der Typ der Nachricht ist immer Acknowledgement.
- Die Tokenlänge ist bei acht Bytes und wird zufällig generiert.
- Der Code ist CREATED (2.01).
- Die MessageId wird zufällig generiert.
- Es werden x-mal Options vom Typ UriPath erstellt.
- Die Payload wird zufällig generiert und ist y Bytes lang.

Die CoAP-Nachricht wird für jeden Durchgang neu generiert und jedem einzelnen Benchmark übergeben.

3.3 Messaufbau

Die Messungen werden auf einem Rechner mit AMD Ryzen 5 2600 (6 Kerne und 12 Threads) als CPU und mit einem Arbeitsspeicher von 16 GB durchgeführt.

Die Netzwerkübertragung findet lokal statt - sprich über die Adresse 127.0.0.1 (Loopback / localhost). Mit dem Kommandozeilenbefehl `start /affinity 1 Server.exe` wird der Server nur auf einem einzelnen Kern ausgeführt, damit nur die reine Leistung des Servers betrachtet wird und nicht durch das Scheduling des Rechners bzw. der CPU verfälscht wird.

Für das Szenario der Serialisierung und Deserialisierung werden keine speziellen Einstellungen vorgenommen, da hier die Standardeinstellungen von BenchmarkDotnet verwendet werden.

3.4 Messergebnisse (Nachrichtenübertragung)

Für die Nachrichtenübertragung wurde die Anzahl der Options auf 100 und die Größe der Payload auf 100000 limitiert. Größere Werte haben keine bemerkenswerte Erkenntnis gebracht und wurde zwecks Übersichtlichkeit weggelassen.

Start	Ende	Laufzeit
2021-11-11T10:17:31.0524370+01:00	2021-11-11T10:19:16.5499204+01:00	00:01:45.4974834
2021-11-11T10:22:12.9707397+01:00	2021-11-11T10:23:58.8623836+01:00	00:01:45.8916439
2021-11-11T10:28:32.5175832+01:00	2021-11-11T10:30:18.3405198+01:00	00:01:45.8229366
2021-11-11T10:32:01.7660773+01:00	2021-11-11T10:33:47.4505300+01:00	00:01:45.6844527
2021-11-11T10:35:34.7139187+01:00	2021-11-11T10:37:20.3576467+01:00	00:01:45.6437280

Tabelle 2: Asynchrone Übertragung mit 100 Options und mit einer Payload von 100 Bytes.

Berechnet man nun die durchschnittliche Laufzeit aus den Ergebnissen in Tabelle 2, in der eine CoAP-Nachricht mit 100 Options und einer Payload von 100 Bytes übermittelt wurde, mittels folgender Formel. Diese Formel wird auch für die nachfolgenden Messungen verwendet, um deren durchschnittliche Laufzeit zu ermitteln.

$$t_{durchschnitt} = \frac{01{:}45.4974834}{5} + \frac{01{:}45.8916439}{5} + \frac{01{:}45.8229366}{5} + \\ + \frac{01{:}45.6844527}{5} + \frac{01{:}45.6437280}{5} = 105,71\text{Sekunden} \tag{1}$$

Daraus ergibt sich für die asynchrone Übertragung eine durchschnittliche Laufzeit von 105,71 Sekunden. Wird diese nun auch für die Messdaten in Tabelle 3 berechnet, ergibt sich hier eine durchschnittliche Laufzeit von 105,61 Sekunden.

Damit ist die synchrone Übertragung um 100 Millisekunden schneller als die asynchrone Übertragung. Durch die geringe Datenmenge von 25813 Bytes für die gesamte Nachricht ist dies nicht überraschend. Bei der synchronen Übertragung wird so lange gewartet, bis die komplette CoAP-Nachricht übertragen wurde. Im Gegensatz dazu wird bei der asynchronen Übertragung die Daten sofort verarbeitet, wenn diese verfügbar sind. Da jedoch immer nur vier Bytes pro Sekunde versendet werden, ist der Mehraufwand zum Erzeugen der asynchronen Zustandsmaschine zu groß und damit unvorteilhaft für die Performanz.

Start	Ende	Laufzeit
2021-11-11T10:17:31.0525437+01:00	2021-11-11T10:19:16.0534992+01:00	00:01:45.0009555
2021-11-11T10:22:12.9707397+01:00	2021-11-11T10:23:58.8617438+01:00	00:01:45.8910041
2021-11-11T10:28:32.5175895+01:00	2021-11-11T10:30:18.3389314+01:00	00:01:45.8213419
2021-11-11T10:32:01.7660713+01:00	2021-11-11T10:33:47.4500247+01:00	00:01:45.6839534
2021-11-11T10:35:34.7139140+01:00	2021-11-11T10:37:20.3586945+01:00	00:01:45.6447805

Tabelle 3: Synchrone Übertragung mit 100 Options und mit einer Payload von 100 Bytes.

Entnimmt man die Messdaten aus der Tabelle 4, die einen Messdurchlauf für eine asynchrone Übertragung einer CoAP-Nachricht mit 100 Options und einer Payload von 1000 Bytes darstellt, ergibt sich eine durchschnittliche Laufzeit von 341,01 Sekunden.

Start	Ende	Laufzeit
2021-11-11T11:53:36.3360380+01:00	2021-11-11T11:59:17.2094861+01:00	00:05:40.8734481
2021-11-11T11:25:13.6291350+01:00	2021-11-11T11:30:54.8399509+01:00	00:05:41.2108159
2021-11-11T11:29:42.9196146+01:00	2021-11-11T11:35:24.2404882+01:00	00:05:41.3208736
2021-11-11T11:32:06.6750943+01:00	2021-11-11T11:37:47.7269970+01:00	00:05:41.0519027
2021-11-11T11:33:33.8796230+01:00	2021-11-11T11:39:14.4538810+01:00	00:05:40.5742580

Tabelle 4: Asynchrone Übertragung mit 100 Options und mit einer Payload von 1000 Bytes

Für eine synchrone Übertragung derselben Nachricht, Messdaten aus Tabelle 5 entnommen, ergibt sich eine durchschnittliche Laufzeit von 341,00 Sekunden. Damit beträgt der Abstand zwischen synchron und asynchron nur 10 Millisekunden.

Start	Ende	Laufzeit
2021-11-11T11:53:36.3360380+01:00	2021-11-11T11:59:17.2084637+01:00	00:05:40.8724257
2021-11-11T11:25:13.6291344+01:00	2021-11-11T11:30:54.8195384+01:00	00:05:41.1904040
2021-11-11T11:29:42.9196145+01:00	2021-11-11T11:35:24.2416850+01:00	00:05:41.3220705
2021-11-11T11:32:06.6750956+01:00	2021-11-11T11:37:47.7273639+01:00	00:05:41.0522683
2021-11-11T11:33:33.8796186+01:00	2021-11-11T11:39:14.4542465+01:00	00:05:40.5746279

Tabelle 5: Synchrone Übertragung mit 100 Options und mit einer Payload von 1000 Bytes

Erhöht man die Payload auf 10000 Bytes, ergibt sich für eine asynchrone Übertragung eine durchschnittliche Laufzeit von 2688,88 Sekunden und für eine synchrone Übertragung eine durchschnittliche Laufzeit von 2688,95 Sekunden. Die jeweiligen Daten wurden dabei von den Tabellen 6 und 7 entnommen. Damit erhöht sich der Abstand auf 70 Millisekunden, das den Trend entspricht, dass mit steigender Payloadgröße, der Vorteil der Asynchronität tragender wird.

Start	Ende	Laufzeit
2021-11-11T11:44:51.4637384+01:00	2021-11-11T12:29:39.2904876+01:00	00:44:47.8267492
2021-11-11T11:45:48.8515388+01:00	2021-11-11T12:30:46.9714652+01:00	00:44:58.1199264
2021-11-11T11:46:46.0080509+01:00	2021-11-11T12:31:33.3480246+01:00	00:44:47.3399737
2021-11-11T11:47:52.2395272+01:00	2021-11-11T12:32:38.3763629+01:00	00:44:46.1368357
2021-11-11T11:49:06.5417421+01:00	2021-11-11T12:33:51.5103142+01:00	00:44:44.9685721

Tabelle 6: Asynchrone Übertragung mit 100 Options und einer Payload von 10000 Bytes

Start	Ende	Laufzeit
2021-11-11T11:44:51.4637476+01:00	2021-11-11T12:29:39.6267910+01:00	00:44:48.1630434
2021-11-11T11:45:48.8515384+01:00	2021-11-11T12:30:46.9709319+01:00	00:44:58.1193935
2021-11-11T11:46:46.0080501+01:00	2021-11-11T12:31:33.3486035+01:00	00:44:47.3405534
2021-11-11T11:47:52.2395282+01:00	2021-11-11T12:32:38.3767620+01:00	00:44:46.1372338
2021-11-11T11:49:06.5417372+01:00	2021-11-11T12:33:51.5109177+01:00	00:44:44.9691805

Tabelle 7: Synchrone Übertragung mit 100 Options und mit einer Payload von 10000 Bytes

3.5 Messergebnisse (Deserialisierung und Serialisierung)

Die nachfolgenden Ergebnisse spiegeln die Ergebnisse der jeweiligen synchronen und asynchronen Deserialisierungs- und Serialisierungs-Methoden wieder. Dabei wurde BenchmarkDotnet verwendet, um diese vier Methoden zu testen.

Die angeführten Tabellen sind die Ergebnisse die durch BenchmarkDotnet ermittelt worden sind. Dabei ergibt sich folgende Legende:

- Method: Der Name der zu testenden Methode.
- AmountOfOptions: Anzahl der Options (in diesem Fall Options des Typs UriPath).
- LengthOfPayload: Länge der Payload in Bytes.
- Mean: Arithmetisches Mittel aus allen Messungen.
- Error: Die Hälfte des 99,9%-igen Konfidenzintervalls.
- StdDev: Die Standardabweichung aller Messungen.
- Gen X: Anzahl der Garbage Collector Generation X Sammlungen jede 1000 Operationen.
- Allocated: Größe des verwalteten (*managed*) Speichers.

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Allocated
SerializeAsync	0	0	5.695 s	0.0402 s	0.0336 s	0.0839	376 B
Serialize	0	0	5.094 s	0.1006 s	0.1198 s	0.0687	288 B
DeserializeAsync	0	0	2.116 s	0.0406 s	0.0791 s	0.3777	1,584 B
Deserialize	0	0	1.639 s	0.0312 s	0.0347 s	0.3948	1,656 B

Tabelle 8: Benchmark mit 0 Options und mit einer Payload von 0 Bytes

Ist nur der Header und der Token in der Nachricht enthalten, wie in Tabelle 8 zusehen, dann sind sowohl asynchron als auch synchron gleichauf. Der Unterschied in der durchschnittlichen Laufzeit zwischen asynchron und synchron ist dem Overhead der

asynchronen Zustandsmaschine, die vom C#-Compiler generiert wird, geschuldet. Auch der leicht erhöhte Speicherverbrauch lässt sich darauf zurückführen.

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Allocated
SerializeAsync	0	1000	5.914 s	0.0909 s	0.0805 s	0.0839	376 B
Serialize	0	1000	6.207 s	0.1199 s	0.1177 s	0.3128	1,312 B
DeserializeAsync	0	1000	3.284 s	0.0636 s	0.0732 s	0.9232	3,824 B
Deserialize	0	1000	3.261 s	0.0631 s	0.0727 s	0.9308	3,864 B

Tabelle 9: Benchmark mit 0 Options und mit einer Payload von 1000 Bytes

Vergrößert man jedoch die Größe der Payload, wie in Tabelle 9 ersichtlich, zeigt sich, dass der Abstand geringer wird. Auch ist anzumerken, dass *SerializeAsync* nun schneller ist als *Serialize*. Waren es zuvor 0,6 Sekunden Unterschied zwischen *Serialize* und *SerializeAsync*, sind es nun 0,293 Sekunden. Bei *Deserialize* und *DeserializeAsync* betrug die Zeitdifferenz noch 0,477 Sekunden, verringert sich diese Differenz bei diesem Durchlauf auf 0,023 Sekunden.

Der Speicherverbrauch blieb bei *SerializeAsync* unverändert, im Gegensatz zu *Serialize* mit einer Zunahme des allokierten Speichers um 1024 Bytes. Bei beiden Methoden für das Serialisieren von CoAP-Nachrichten hat sich der Speicherverbrauch zwar auch erhöht, jedoch blieb der Abstand zueinander unverändert.

Eine mögliche Erklärung für den gleichbleibenden Speicherverbrauch von *SerializeAsync* ist, dass dieser mittels eines *Stream* arbeitet, der fortlaufend beschrieben wird. Im Gegensatz dazu verwendet *Serialize* intern einen die *PooledMemoryBufferWriter*-Klasse. Dieser stellt Methoden bereit, um auf einem Puffer, bestehend aus einer Menge von `Memory<byte>`s, zu schreiben. Dabei muss der Aufrufer nur Speicher vom *PooledMemoryBufferWriter äusleihen"*, diesen mit den gewünschten Daten beschreiben und dem *PooledMemoryBufferWriter* die Anzahl der geschriebenen Bytes mitteilen, damit die Position des *PooledMemoryBufferWriter*s weitergeschoben werden kann. Dies wird im Codebeispiel 6 veranschaulicht.

Der Nachteil des *PooledMemoryBufferWriter*s ist, dass dieser seinen zur Verfügung stehenden Speicherbereich vergrößern muss, wenn die Kapazität erschöpft ist. Dabei vergrößert sich dieser so weit, damit der Puffer die zu schreibenden Daten aufnehmen kann. Auch ist der *PooledMemoryBufferWriter* eine Eigenimplementierung, jedoch wurde im Laufe der Recherchen für diese Arbeit eine ähnliche Implementation von Microsoft[5] gefunden, die dieses Problem möglicherweise besser handhabt, als die derzeitige Implementation. Da dies jedoch einen zu großen Aufwand darstellt, wurde darauf verzichtet.

Wird die Payload weiter vergrößert, in diesem Fall auf 100000 Bytes (100 kB), sieht man, dass die asynchronen Methoden deutlich schneller sind als die synchronen Methoden. Dies resultiert darin, dass bei I/O-lastigen Aufgaben, in diesem Fall das Lesen bzw. Schreiben der Payload vom bzw. auf den `Stream` oder dem `ReadonlyMemory<byte>`, die

[5]Dokumentation des MemoryBufferWriter

```
// Erzeugung des PooledMemoryBufferWriters.
using (var writer = new PooledMemoryBufferWriter())
{
    // Anfordern eines Memory<byte> in der Größe von 2048 Bytes.
    var buffer = writer.GetMemory(2048);

    // Erzeugung eines Zufallgenerators.
    var random = new Random();

    // Befüllung des Puffers mit zufälligen Werten.
    random.NextBytes(buffer.Span);

    // Benachrichtung des Writers, dass der gesamte Puffer beschrieben wurde.
    writer.Advance(buffer.Length);
}
```

Listing 6: Verwendung des PooledMemoryBufferWriters

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SerializeAsync	0	100000	8.611 s	0.1519 s	0.1268 s	0.0763	-	-	376 B
Serialize	0	100000	84.259 s	1.1186 s	1.0464 s	31.1279	31.1279	31.1279	100,312 B
DeserializeAsync	0	100000	57.500 s	1.1007 s	1.2675 s	83.3130	83.3130	83.3130	264,040 B
Deserialize	0	100000	107.972 s	1.3266 s	1.1078 s	83.2520	83.2520	83.2520	263,984 B

Tabelle 10: Benchmark mit 0 Options und mit einer Payload von 100000 Bytes

Asynchronität ihre Vorteile ausspielen kann, da eine große Menge an Daten gelesen oder geschrieben wird. Andere I/O-lastige Aufgaben sind etwa die Übertragung von Daten über das Netzwerk oder jeglicher Zugriff auf das Filesystem. Es wird dabei asynchron die Daten vom jeweiligen Stream gelesen (siehe Codebeispiel 7) bzw. auf den jeweiligen Stream geschrieben (siehe Codebeispiel 8).

Sobald sich jedoch die Anzahl der Options erhöht, sinkt die durchschnittliche Laufzeit der asynchronen Methoden und der Abstand zu den synchronen Methoden vergrößert sich (siehe Tabelle 11). Hierbei steigt die Laufzeit von *SerializeAsync* auf 2191 Sekunden und der Speicherverbrauch auf 5,383 KB. Im Gegensatz dazu schneidet *Serialize* mit 325 Sekunden und einem Speicherverbrauch von 160 KB deutlich besser ab. Da dies eine ungewöhnliche Abweichung darstellt, wurde nachgeforscht und der Grund für die Verlangsamung von *SerializeAsync* gefunden: Um Options, die einen *string* als Wert besitzen, auf einen Stream zu schreiben, wurde die *StreamWriter*-Klasse verwendet. Diese ermöglicht es, mit einer bestimmten Zeichenkodierung auf einen *Stream* zu schreiben (siehe Codebeispiel 9).

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Allocated
SerializeAsync	1000	0	2,191.0 s	42.42 s	56.63 s	1316.4063	-	5,383 KB
Serialize	1000	0	325.0 s	3.36 s	2.98 s	39.0625	-	160 KB
DeserializeAsync	1000	0	950.3 s	11.10 s	9.84 s	122.0703	37.1094	526 KB
Deserialize	1000	0	645.3 s	12.27 s	12.05 s	74.2188	24.4141	323 KB

Tabelle 11: Benchmark mit 1000 Options und mit einer Payload von 0 Bytes, jedoch mit StreamWriter

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Allocated
SerializeAsync	1000	0	740.2 s	14.14 s	15.72 s	50.7813	-	211 KB
Serialize	1000	0	330.7 s	6.57 s	5.82 s	39.0625	-	160 KB
DeserializeAsync	1000	0	952.6 s	16.95 s	15.03 s	121.0938	40.0391	526 KB
Deserialize	1000	0	591.0 s	8.91 s	8.33 s	74.2188	24.4141	323 KB

Tabelle 12: Benchmark mit 1000 Options und mit einer Payload von 0 Bytes, jedoch ohne StreamWriter.

Verwendet man jedoch die angebotenen Schreibmethoden von der *Stream*-Klasse, indem man zuvor den String durch die entsprechende Zeichenkodierung in *bytes* umwandeln lässt (siehe Codebeispiel 10), verringert sich die durchschnittliche Laufzeit und auch der Speicherverbrauch (siehe Tabelle 12).

Eine mögliche Erklärung, warum bei der Verwendung des *StreamWriter*s eine so hohe Laufzeit und Speicherverbrauch anfällt, kann leider nicht erbracht werden, da hier das Wissen des Verfassers übersteigt. Vergleicht man den Code der jeweiligen *WriteAsync*-Methoden von *StreamWriter*[6] mit der von *Stream*[7], kann man einige Optimierungen bei *Stream* erkennen, die wahrscheinlich einen Unterschied ausmachen.

Die nachfolgenden Messungen wurden ohne die Nutzung des *StreamWriter*s durchgeführt. Dies resultierte darin, dass die durchschnittliche Laufzeit von *SerializeAsync* sich auf einem niedrigeren Niveau eingependelt hat, im Vergleich zu den Messungen, in denen noch der *StreamWriter* verwendet wurde.

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Allocated
SerializeAsync	1000	1000	715.4 s	14.28 s	15.87 s	50.7813	-	211 KB
Serialize	1000	1000	315.6 s	1.90 s	1.78 s	39.0625	-	161 KB
DeserializeAsync	1000	1000	923.6 s	6.20 s	5.18 s	121.0938	41.0156	529 KB
Deserialize	1000	1000	593.0 s	5.02 s	4.45 s	75.1953	15.6250	326 KB

Tabelle 13: Benchmark mit 1000 Options und mit einer Payload von 1000 Bytes

Ab dem Zeitpunkt, in dem sich eine größere Anzahl an Options in der CoAP-Nachricht befinden, können die asynchronen Methoden ihren Geschwindigkeitsvorteil vom Szenario, in dem nur eine Payload mit weniger als 100000 Bytes (100 KB) gegeben war (siehe

[6]Implementierung von StreamWriter auf GitHub
[7]Implementierung von Stream auf Github

Tabellen 8, 9 und 10), nicht mehr ausspielen. Dies ist darauf zurückzuführen, dass bei kleineren Datenmengen der Overhead, der durch die asynchrone Zustandsmaschine erzeugt wird, zu stark überwiegt. Deshalb werden auf die Ergebnisse, visualisiert durch die Tabellen 14 bis 17, nicht näher eingegangen.

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SerializeAsync	1000	100000	721.9 s	13.99 s	24.86 s	50.7813	-	-	211 KB
Serialize	1000	100000	368.6 s	6.78 s	6.34 s	64.4531	32.2266	32.2266	258 KB
DeserializeAsync	1000	100000	1,047.4 s	13.92 s	12.34 s	164.0625	82.0313	82.0313	783 KB
Deserialize	1000	100000	767.0 s	15.31 s	16.38 s	83.0078	83.0078	83.0078	580 KB

Tabelle 14: Benchmark mit 1000 Options und mit einer Payload von 100000 Bytes

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SerializeAsync	100000	0	68.01 ms	0.651 ms	0.609 ms	5125.0000	-	-	21 MB
Serialize	100000	0	30.87 ms	0.320 ms	0.283 ms	3843.7500	31.2500	31.2500	16 MB
DeserializeAsync	100000	0	144.43 ms	2.678 ms	2.505 ms	8500.0000	3250.0000	1000.0000	50 MB
Deserialize	100000	0	116.40 ms	2.222 ms	5.281 ms	5400.0000	2000.0000	800.0000	32 MB

Tabelle 15: Benchmark mit 100000 Options und mit einer Payload von 0 Bytes

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SerializeAsync	100000	1000	69.35 ms	0.878 ms	0.778 ms	5125.0000	-	-	21 MB
Serialize	100000	1000	31.50 ms	0.406 ms	0.339 ms	3843.7500	31.2500	31.2500	16 MB
DeserializeAsync	100000	1000	152.99 ms	3.035 ms	7.154 ms	8500.0000	3250.0000	1000.0000	50 MB
Deserialize	100000	1000	110.34 ms	2.188 ms	4.320 ms	5800.0000	2400.0000	1000.0000	32 MB

Tabelle 16: Benchmark mit 100000 Options und 1000 Bytes

Vergrößert man jedoch die Größe der Payload über 100 MB (in Tabelle 19 auf 1 GB), wird der Abstand zwischen *Serialize* und *SerializeAsync* sowohl im Hinblick auf die durchschnittliche Laufzeit als auch den Speicherverbrauch deutlich größer. Hierbei ist *SerializeAsync* um den Faktor 7 schneller im Durchschnitt und um den Faktor 143 effizienter im Speicherverbrauch.

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SerializeAsync	100000	100000	69.25 ms	0.458 ms	0.383 ms	5125.0000	-	-	21 MB
Serialize	100000	100000	32.23 ms	0.619 ms	0.826 ms	3843.7500	31.2500	31.2500	16 MB
DeserializeAsync	100000	100000	166.15 ms	4.591 ms	13.391 ms	8750.0000	3250.0000	1000.0000	50 MB
Deserialize	100000	100000	113.15 ms	2.256 ms	6.545 ms	5400.0000	2000.0000	800.0000	32 MB

Tabelle 17: Benchmark mit 100000 Options und mit einer Payload von 100000 Bytes (100 KB)

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SerializeAsync	100000	1000000	69.00 ms	1.069 ms	0.948 ms	5125.0000	-	-	21 MB
Serialize	100000	1000000	34.34 ms	0.407 ms	0.361 ms	4062.5000	250.0000	250.0000	19 MB
DeserializeAsync	100000	1000000	140.89 ms	1.490 ms	1.394 ms	8500.0000	3250.0000	750.0000	52 MB
Deserialize	100000	1000000	102.68 ms	2.028 ms	2.082 ms	5400.0000	2000.0000	800.0000	34 MB

Tabelle 18: Benchmark mit 100000 Options und mit einer Payload von 1000000 Bytes (100 MB)

Method	AmountOfOptions	LengthOfPayload	Mean	Error	StdDev	Gen 0	Gen 1	Allocated
SerializeAsync	100000	1000000000	158.1 ms	3.13 ms	5.06 ms	5000.0000	-	21 MB
Serialize	100000	1000000000	1,112.3 ms	21.24 ms	25.29 ms	3000.0000	-	3,016 MB
DeserializeAsync	100000	1000000000	1,507.7 ms	28.40 ms	29.16 ms	7000.0000	2000.0000	4,003 MB
Deserialize	100000	1000000000	1,764.1 ms	34.93 ms	52.28 ms	4000.0000	1000.0000	2,893 MB

Tabelle 19: Benchmark mit 100000 Options und mit einer Payload von 1000000000 Bytes (1 GB)

```
1  public class StreamReader
2  {
3      private readonly IBufferWriter writer;
4
5      public StreamReader(IBufferWriter writer)
6      {
7          this.writer = writer;
8      }
9
10     public async Task<byte[]> ReadAsync(Stream stream, CancellationToken ct)
11     {
12         while (true)
13         {
14             // Ausleihen eines Speicherbereichs als Puffer.
15             var buffer = this.writer.GetMemory(2048);
16
17             // Daten vom Stream lesen.
18             var bytesRead = await stream.ReadAsync(buffer, ct)
19             .ConfigureAwait(false);
20
21             // Keinen Daten mehr im Stream.
22             if (bytesRead == -1)
23             {
24                 // Lesevorgang beenden.
25                 break;
26             }
27
28             // Position des IBufferWriters verschieben.
29             this.writer.Advance(bytesRead);
30         }
31
32         // Rückgabe alles Daten, die vom Stream gelesen wurden.
33         return this.writer.WrittenMemory.ToArray();
34     }
35 }
```

Listing 7: Asynchrones Lesen eines Streams mittels IBufferWriters

```
public class StreamWriter
{
    public async Task WriteAsync(
        Stream stream,
        ReadonlyMemory<byte> value,
        CancellationToken ct)
    {
        await stream.WriteAsync(value, ct).ConfigureAwait(false);
    }
}
```

Listing 8: Asynchrones Beschreiben eines Streams

```
public async Task WriteAsync(Stream stream, string value, CancellationToken ct)
{
    await using (var writer = new StreamWriter(stream, Encoding.UTF8 4096, true))
    {
        await writer.WriteAsync(value.AsMemory(), ct).ConfigureAwait(false);
    }
}
```

Listing 9: Asynchrones Schreiben eines strings auf einen Stream mittels StreamWriter

```
public async Task WriteAsync(Stream stream, string value, CancellationToken ct)
{
    var bytes = Encoding.UTF8.GetBytes(value);
    await stream.WriteAsync(bytes, ct).ConfigureAwait(false);
}
```

Listing 10: Asynchrones Schreiben eines strings auf einen Stream

4 Diskussion der Messergebnisse

Betrachtet man die dargelegten Messergebnisse in Kapitel 3, sieht man, dass bei großen und aufwendigen I/O-Operationen Asynchronität besser abschneidet. Im Gegensatz dazu schlagen sich synchrone Methoden besser, wenn sich Menge der Daten im reservierten Speicher, also kein Nachladen oder Anforderung von weiteren Daten, sofort verarbeiten lässt. Dies lässt sich damit argumentieren, dass, wenn sich alle Daten im Speicher befinden, die synchrone Methode / Programm die Daten ohne weiteren Aufwand verwenden kann. Hingegen bei asynchronen Methoden ist es unvorteilhaft, wenn die Daten sehr klein sind und sich somit der Mehraufwand zum Aufbau der dafür benötigten Zustandsmaschine nicht lohnt.

Dies hat sich in beiden Messszenarien gezeigt, jedoch erst ab einer sehr großen Menge an Daten. Somit profitiert CoAP von Asynchronität, wenn eine große Menge an Daten verarbeitet werden müssen. Ist jedoch die zu verarbeitende Menge an Daten klein, überwiegt der Mehraufwand der asynchronen Zustandsmaschine. Somit wirkt sich die Asynchronität bei kleineren Datenmengen, in diesem Messszenario kleiner als 100000 Bytes, negativ auf die Performanz der CoAP-Bibliothek aus.

Um diesen negativen Effekt entgegenzuwirken, gehen auch viele Entwickler von Softwareprogrammen, die gewisse asynchrone Methoden anbieten bzw. verwenden, dazu über, anhand von bestimmten Bedingungen oder Kriterien zu ermitteln, ob eine asynchrone oder eine synchrone Variante der zu implementierenden Funktion verwendet werden soll. Damit wird versucht eine asynchrone Methode dem Entwickler zur Verfügung zu stellen, die optimiert auf die jeweiligen Parameter ist und somit in jeglichen Fall die bestmögliche Leistung erbringt.

Jedoch muss der Entwickler abwägen, ob Nachrichten mit solch einer großen Payload innerhalb der Softwareapplikation zur Norm gehören. Anzumerken ist, dass sowohl die synchrone als auch asynchrone Implementierung Raum für Optimierungen offen lässt. Diese sind auch Gegenstand von weiteren Maßnahmen, die man im Rahmen dieses Projekts vornehmen kann. Auf diese werden jedoch im Kapitel 5 näher eingegangen.

5 Schlussfolgerung

Hat nun die asynchrone Implementation eines Servers, in diesem Falle des *Serializer*s, einen Einfluss auf dessen Durchsatzrate? Diese Frage kann nicht eindeutig beantwortet werden. Diese ist abhängig davon, wie man folgende Fragen beantwortet:

- Ist die Menge der zu verarbeitenden Daten klein? Wenn ja, dann hat Asynchronität keine große Auswirkung, sondern erzeugt eine erhebliche Mehrarbeit, aufgrund der zu erzeugenden Zustandsmaschine.
- Ist die Menge der zu verarbeitenden Daten groß? Wenn ja, dann hat Asynchronität eine große Auswirkung auf den Durchsatz, da durch die kürzere Verarbeitungszeit schneller neu ankommende Daten verarbeitet werden können.

Sieht man sich die Ergebnisse im Kapitel 3 an, bemerkt man, dass diese nicht eindeutig für Asynchronität oder Synchronität sprechen. Wenn keine CoAP-Options enthalten sind (siehe Tabellen 8 bis 10), dann verringert sich der Abstand von den asynchronen zu den synchronen Methoden mit größer werdender Payload.

Ab dem Zeitpunkt, in dem CoAP-Options in die Nachricht eingefügt werden, ist die synchrone Variante immer schneller als die asynchrone. Dadurch dass die Menge an Bytes, die für eine einzelne Option benötigt wird, sehr gering ist, hat *SerializeAsync* bzw. *DeserializeAsync* immer den Mehraufwand für den Aufbau seiner asynchronen Zustandsmaschine.

Erst mit einer sehr großen Payload von 1 GB zeichnet sich ein deutliches Ergebnis pro Asynchronität ab - zumindest bei der Verwendung der *SerializeAsync*-Methode. Hier ist *SerializeAsync*, wie schon festgestellt worden ist, bei der durchschnittlichen Laufzeit um den Faktor 7 schneller und um den Faktor 143 effizienter im Speicherverbrauch als die *Serialize*-Methode.

Was bedeutet dies nun für die, in dieser Arbeit gestellten Ausgangsfrage? Eine eingebaute Asynchronität in Softwareprogrammen ist nicht für jedes Problem die passende Lösung. Es kann seine Vorteile gut in Situationen ausspielen, in denen auf ein Ergebnis einer zeit- oder rechenintensiven Aktion gewartet werden muss. Diese können in Form von Netzwerkübertragungen (Anfragen bzw. Antworten mittels HTTP, Datenbankabfragen), rechenintensiven Berechnungen oder speicherintensiven I/O-Vorgängen (Lesen einer großen Datei von der Festplatte) auftreten. Für einfache und schnell auszuführende Aufgaben ist eine asynchrone Methode eher die falsche Wahl, da wie schon erwähnt, der Aufwand der asynchronen Zustandsmaschine zu groß wird.

Darum ist es abhängig von den Anforderungen und Aufgabenstellung der zu entwickelnden Software. Interagiert der betreffende Codeteil mit externen Systemen (Datenbanksystemen, REST-APIs, Internetservern) oder führt dieser rechenintensive Berechnungen durch, dann ist das Verwenden von Asynchronität zu empfehlen. Damit wird der Main-Thread entlastet, was dazu führt, dass bei Applikationen ohne grafisches Interface (GUI)

die Auslastung des Threadpools reduziert wird und bei Applikationen mit GUI der Main-Thread nicht blockiert und somit die Oberfläche benutzbar bleibt.

Betrachtet man nun den Fall eines Servers, dann kann man dazu tendieren auf eine asynchrone Implementierung zu setzen, da hier Daten über ein beliebiges Medium, wie zum Beispiel Ethernet oder WLAN, und Protokoll an den Server geschickt werden. Dabei weiß der Server jedoch nicht im Vorhinein, wann diese Übertragung komplett abgeschlossen ist. Um nicht den Main-Thread des Servers zu blockieren, wie es bei einem synchronen Server passieren würde, kann durch den Einsatz von Asynchronität der Main-Thread entlastet und somit der Durchsatz gesteigert werden. Vor allem im speziellen Fall, wenn eine große Menge an Daten (siehe Tabelle 19) verarbeitet werden muss, kann die Asynchronität eines Programms deutliche Performanz- und Speichervorteile bringen. Nur durch die Verwendung von Asynchronität können sieben CoAP-Nachrichten, mit einer großen Anzahl an Options (100000) und einer großen Payload (1 GB), in der gleichen Zeit verarbeitet werden, wie als wenn man die synchrone *Serialize*-Methode verwendet.

Ist dies jedoch ein triftiger Grund, um nur asynchrone Programme zu schreiben? Nicht unbedingt. Es kommt auf die gestellten Anforderungen an die Software an, ob diese sehr performant und ressourcenschonend arbeiten sollte. Ist dies der Fall, dann sollte ein großes Augenmerk darauf gelegt werden, zu überlegen, an welchen Stellen im Programm eine asynchrone gegenüber einer synchronen Methode vorgezogen werden sollte. Hierbei bieten viele Bibliotheken, zumindest im C#-Ökosystem, sowohl synchrone als auch asynchrone Methoden an, die genau dieselbe Funktion abbilden, jedoch abhängig davon, ob es synchron oder asynchron passieren soll. Ist man jedoch selbst ein Entwickler einer solchen Bibliothek, ist es schwierig abzuwägen, welche Methoden einer asynchronen oder einer synchronen Implementierung benötigen. Betrachtet man die Referenzquelle[8] von Microsoft für .NET an, sieht man, dass die Entwickler dazu tendieren, anhand von verschiedenen Parametern zu entscheiden, ob die darunterliegende Funktion synchron oder asynchron ausgeführt werden sollte. Ein fiktives Beispiel dafür wird im Codebeispiel 11 aufgeführt.

Für das CoAP-Protokoll, spezifiziert nach RFC 7252 von Shelby, Hartke und Bormann [3], erzielt Asynchronität keine bemerkenswerten Verbesserungen, da durch die Größenlimitierung von UDP-Paketen (maximal 65535 Bytes) die Datenmenge zu klein ist. Implementiert man die Funktion des blockweisen Datentransfers, wie im RFC 7959 von Shelby und Bormann [2] beschrieben, könnte diese Limitierung umgangen werden und größere Datengrößen erzielt werden. Man kann sich auch dazu entscheiden auf die neuere Standardisierung, wie im RFC 7959 von Shelby und Bormann beschrieben [2], zu setzen, um größere Mengen an Daten, ohne großen Aufwand, zu übertragen.

Das Fazit aus dieser Arbeit ist damit, das der Vorteil von asynchronen Server bzw. Programmen sehr davon abhängt, in welchen Szenarien diese eingesetzt werden. Müssen

[8] Referenzquelle von Microsoft zu .NET

```
public async Task<byte[]> ReadAsync(Stream stream, CancellationToken ct)
{
    // Ausleihen eines Speichers.
    var owner = MemoryPool<byte>.Shared.Rent(2048);
    var buffer = owner.Memory;

    // Überprüfen, ob der gesamte Stream in den Puffer geladen werden kann.
    if (stream.Length <= 2048)
    {
        // Lesen des Speichers in synchroner Variante.
        var bytesRead = stream.Read(buffer.Span);
        return buffer.ToArray();
    }

    // Sonst asynchrones Lesen des gesamten Streams.
    // Das passiert folgendermaßen:
    // 1) Lesen des Streams mittels ReadAsync.
    // 2) Überprüfen ob Ende des Streams erreicht wurde.
    // 2.1) Wenn ja --> den Inhalt des Puffers zurückgeben.
    // 2.2) Wenn nein --> Puffer verdoppeln und mit Punkt 1 beginnen.
}
```

Listing 11: Optimierte asynchrone Methode

viele Anfragen mit großer Datenmenge gleichzeitig verarbeitet werden, dann ist ein asynchroner Server einem synchronen vorzuziehen. Dies wird dadurch noch verstärkt, dass eine asynchrone Implementierung eines Servers, zumindest in C#, durch eine einfache und durchdachte Syntax bestehend aus `async`, `await` und `Task` bzw. `Task<TResult>` ermöglicht wird. Auch ist der Trend bemerkbar, dass viele bekannte Bibliotheken eine große Anzahl an asynchronen Methoden anbieten, damit diese auch in asynchronen Konstrukten verwendet werden können.

Jedoch wird hier auch eine generelle Empfehlung zur Implementierung und Verwendung von asynchronen Methoden ausgesprochen. Die gemessenen, zeitlichen Unterschiede zwischen Asynchronität und Synchronität befinden sich im vertretbaren Rahmen. Ist man nicht auf vollkommene Optimierung der Performanz der Software fokussiert, ist dieser Unterschied vernachlässigbar.

Literatur

[1] C. Bormann u. a. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*. RFC 8323. RFC Editor, Feb. 2018. URL: http://www.rfc-editor.org/rfc/rfc8323.txt.

[2] Z. Shelby und C. Bormann. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. RFC Editor, Aug. 2016. URL: http://www.rfc-editor.org/rfc/rfc7959.txt.

[3] Z. Shelby, K. Hartke und C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. RFC Editor, Juni 2014. URL: http://www.rfc-editor.org/rfc/rfc7252.txt.