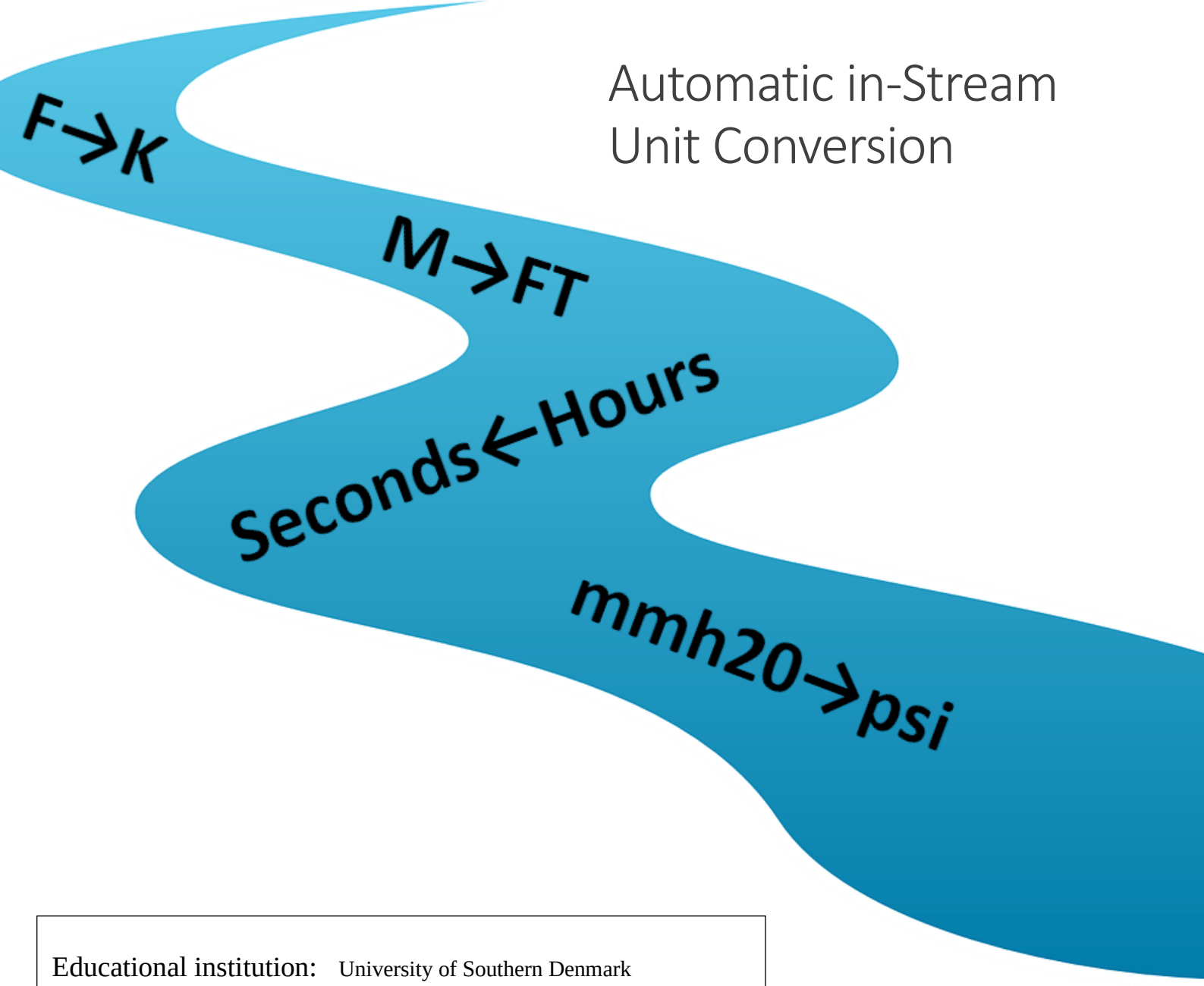
Automatic in-Stream Unit Conversion



Educational institution:	University of Southern Denmark
Faculty:	The Faculty of Engineering
Institute:	The Maersk Mc-Kinney Moller Institute
Education:	Civil Engineer in Software Engineering
Semester:	6
Module code:	T510012101
Advisor:	Aslak Johansen
Group Nr.:	01

Group Members:

Frederik Alexander Hounsvad	frhou18
Peter Andreas Brændgaard	pebra18

1 Abstract

This project considers the problem of converting between units within a stream, with minimal user input. It aims to do this, while being able to combine any units to make the system as flexible as possible. This has been achieved through an iterative process, where three prototypes have been created. The prototypes have been performance tested to gain knowledge of which implementations solved the problem in the most desirable way.
The system is developed using Microsoft's .NET core. Initial data is posted to an MQTT broker, which the system subscribes to. Other systems can then subscribe to the system created in this project and receive data converted to any compatible unit through a websocket.

2 Foreword

The purpose of this document is to detail the findings and considerations we have made during the process of implementing a conversion service for instream conversion of partially arbitrary units, only limited by the base units in the system.

The report itself is targeted at the project advisor, censor, lector as well as our fellow students, and the technical level of the document will therefore be best understood with at least such a level of knowledge.

We would like to thank the people who have been instrumental to the creation of this project, as this project could not have been conceived outside of the academic environment that we are in.
We as a group would like to thank:

- **Aslak Johansen** for always being available and providing exquisite supervision.
- **Previous lecturers** for teaching us all the knowledge required for completing this project.

30/05/2021

X 

Frederik Alexander Hounsvad

Signed by: frederik

30/05/2021

X 

Peter Andreas Brændgaard

Signed by: Peter

3 Table of Contents

4 Figure overview

5 Introduction

This report is the product of a bachelor's project in software engineering at the University of Southern Denmark.
In a world where energy efficiency has become ever more important, not only due to prices, but also the ever-looming issue of global climate change. The need for simulating and monitoring building energy flows and adjusting indoor environmental conditions to the behavioural patterns of its inhabitants, has become a real and actionable choice for companies. These choices impact both workers having their daily routine in these buildings and residents in modern apartment complexes. It will most likely also extend to private housing as the technology progresses and becomes desirable and achievable for individuals.
All this measuring and simulation creates huge amounts of data. These measurements and calculations are usually done in whatever unit is most suited for the specific case the sensor was created for, or in the units best suited for the software and formulas used for simulation, design of the building, and the system to be optimised. All this is, of course, only exacerbated by the differences in what unit systems different countries use. This large variance in use-cases for data, sources of data, and different unit systems creates a significant need for the conversion of these values in a quick and robust fashion. Our project seeks to explore a solution for converting values directly from streams of data, to maximise sensor options and to minimize the impacts of developing and implementing workarounds for the spread of units for the same types of measurements.

5.1 Context

As a part of building automation and energy optimization large arrays of sensors are used, these sensors may measure light levels, temperature, humidity, pressure, CO2 levels and so on. Such data can be gathered in streams with solutions like Kafka and MQTT. Once data goes into the publish subscribe systems it can be easily read by automation and monitoring systems. This is not always directly possible, as some systems may expect temperature in Celsius, some in Fahrenheit and others in Kelvin. This could limit a user in what systems and sensors they could employ for their solutions by the systems and sensors not supporting the units required.

5.2 Problem

The project focuses on the implementation of a prototype system to convert between units in stream, with minimal user input. This requires a solution to the following:

- How to convert "in stream"
- How to limit the need for user input as much as possible
- Which pub sub stream provider to use
- How to convert arbitrary units
- How to parse units
- How to ensure fast conversions.

5.3 Related Work

This section looks at existing structures and techniques behind unit conversion, used by both researchers and in real life applications. This knowledge will be used as a foundation for our decision making during the project.

5.3.1 Definitions and standards for units and conversions.

The most basic level on which units need to be converted would be manual conversion. Here one would refer to sources of conversion formulae, these sources could be governmental bodies such as NIST for the USA or the international standards committee BIPM, where one can find the definition of units and their conversion factors.
These standards give us a foundation for the way units are converted, and serves as a fact checking source for the base units and their conversion factors. However, they do not provide any solutions for automating the conversion of units, nor do they provide solutions for the conversion of compound units.

5.3.2 Conversion of compound units

Conversion of units that are a combination of other units, is not simple when compared with individual unit conversion, as they do not exist within an official lookup table the same way that the base units do. “Conversion of units of measurement” [1] by G.S Novak goes into detail on how to convert compound units from one form to the desired form, as well as performing dimensional analysis of the unit to ensure that the conversion is valid. Dimensional analysis is the act of looking at what a unit is measuring by looking at the individual components of a compound unit. Both are central to the functionality of the solution we desired to create.

5.3.3 Conversion tools and systems

Units and converting between them is a widespread issue, and as such there are many other solutions out there. Ones that we have looked at are QUDT [2] and parts of WolframAlpha[1] which is a “knowledge engine” that also allows for unit conversion.
These systems use the principals of using dimension vectors, conversion factors and offsets to convert between units, in the same fashion as described in “Conversion of units of measurement” [1]. This seems to be the universally recognised way to handle the conversion of compound units, judging from these other sources.

5.4 Approach

We will go about this development through an iterative process of prototyping, finding issues, planning implementations, and repeating. Through this we hope to achieve a good solution within the timespan the project.

5.5 Project Structure

The project contains a wordlist for acronyms or other word definitions and is structured by having a domain analysis covering some introduction into the domain we are working with. This is followed by a requirements section where the requirements for the project have been elicited through a use case diagram. Following this is a section explaining the tools used for our implementation and development. This chapter is placed before design to give insight into some of the tools before the design process, as the tools had an impact on this. After that section we have our section on the design of the final solution, going over the use if dependency injection, units, conversion, and communication. The design is then followed by the section on the implementation details for different iterations of the software prototype. Finally, we have a section about the testing of the software. The report is then trailed by an evaluation, future works section, and conclusion, as well as a bibliography in IEEE style, and at the end the appendices.

[1] https://www.wolframalpha.com/

6 Word List

IoT – Internet of Things.

"The Internet of things (IoT) describes the network of physical objects—"things" — that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet**"** [3]

SI and BIPM – International System of Units

https://www.bipm.org/en/about-us/

Pub Sub – Publish Subscribe

Publish, Subscribe. Used in references to systems that use subscribing to data being published for data transfer.

DTO – Data transfer object

An object which sole purpose is to aggregate data into a single object without containing any logic.

RTC – Real Time Clock

A device that is responsible for keeping time accurately.

ORM - Object-relational mapping

A tool for converting data into objects which can be used by an object-oriented programming language.

RDF – Resource Description Framework

RDF is a standard made to describe metadata using xml in a fashion readable by computers. It is designed by the World Wide Web Consortium.
https://www.w3.org/RDF/

RDFS – Resource Description Framework Schema

RDFS is an extension upon RDF to increase the vocabulary of RDF.

OWL – Web Ontology Language

OWL is a knowledge representation language build upon RDF with very strict semantic rules.

TURTLE – Terse RDF Tripple Language

Turtle is a concrete language specification for how to represent an RDF graph.

7 Domain analysis

7.1 Units and quantities

Units are standards for measurement of a quantity. The units dealt with by the system are all measuring physical quantities. All the units in the system are either base units or derived units.
There are seven base units, one for each base physical quantity. These seven are: time, length, mass, electric current, thermodynamic temperature, amount of substance, and luminous intensity.
There are two defining characteristics of base quantity; they are independent of each other, i.e. a base quantity is not defined in terms of another quantity, and together they make up all other quantities. These base quantities have been decided by the BIPM, and do not necessarily reflect any physical significance, i.e. the base physical quantities are not more central to the world around us than others.

All other quantities can be described as a mixture of one or more base quantities. For example, area is length squared or acceleration is length per time squared.

The most used system of units is the international system of units (SI), which is also the system to define the base quantities. All the base units have been (re)defined to all reflect physical constants in different ways. This ensures that no matter where on the planet or in the universe, one would have the same point of reference, as these constants are supposed to be just that, constant.

However, there are many different systems of unit, or variations of a single quantity. The most well-known system apart from SI is the imperial system, used by the UK and some of its former colonies. Another well-known deviation is the way we measure time. Minutes, hours, days and so on are not SI units. It is only the second which is an official SI unit, even though the others are accepted. It is, therefore, important to be able to convert between these units.

7.1.1 Dimension vector

As all units are made up of these seven base physical quantities, it is possible to describe a unit in relation to which physical quantities they are constructed from. This representation is called a dimension vector. It can look like this; A0E0L1I0M1H0T-2D0. Each letter is a base physical quantity, A is amount of substance, E is electric current etc. Finally, it also includes D for dimensionless for ratios, which technically do not have a unit. The numbers represent the factors of each dimension i.e., 0 means the dimension is not present. A unit for the previously mentioned dimension vector could be m*kg/s^2. The dimension vector can then be used to determine if two units are measuring the same quantity, and thus if they can be converted from one to the other or not.

7.2 Unit Representation

As described previously, the seven base units make up all other units. This means that units can be combined to measure different things. In theory, they can be combined completely arbitrarily. However, not all combinations have any corresponding physical quantity. A combined unit can be represented as a fraction, with 1 or more units in the numerator and 0 or more units in the denominator. The same unit can occur more than once, such as W*s, m^3 and m/s^2. There are different notations to represent units in the denominator e.g., they can be raised to a negative power or separated by a slash. Seeing as a unit can always be represented as a single fraction, it can always be written with only a single slash, where everything left of the slash is the numerator and everything right of the slash is the denominator. Units are then further separated by a multiplication sign which is a dot in

handwriting or an asterisk (*) in typewriting e.g., W*s. Many of the common combined units have gotten their own names to make it easier to use. Newton (N) is for example equal to kg*m/s^2.

7.2.1 Prefixes

To make it easier to work with both big and small numbers, the SI-system has defined a list of prefixes to represent powers of 10 in increments of three above 1000 and below 0.001 and in increments of one otherwise. These can be seen on Figure 1.

Factor	Name	Symbol	Multiplying Factor
10^{24}	yotta	Y	1 000 000 000 000 000 000 000 000
10^{21}	zetta	Z	1 000 000 000 000 000 000 000
10^{18}	exa	E	1 000 000 000 000 000 000
10^{15}	peta	P	1 000 000 000 000 000
10^{12}	tera	T	1 000 000 000 000
10^{9}	giga	G	1 000 000 000
10^{6}	mega	M	1 000 000
10^{3}	kilo	k	1 000
10^{2}	hecto	h	100
10^{1}	deca	da	10
10^{-1}	deci	d	0.1
10^{-2}	centi	c	0.01
10^{-3}	milli	m	0.001
10^{-6}	micro	μ	0.000 001
10^{-9}	nano	n	0.000 000 001
10^{-12}	pico	p	0.000 000 000 001
10^{-15}	femto	f	0.000 000 000 000 001
10^{-18}	atto	a	0.000 000 000 000 000 001
10^{-21}	zepto	z	0.000 000 000 000 000 000 001
10^{-24}	yocto	y	0.000 000 000 000 000 000 000 001

Figure 1: Table 1SI prefixes, Source: https://www.bipm.org/en/measurement-units/prefixes.html

However, these prefixes are not the only used prefixes. In 1998, the International Electrotechnical Commission (IEC) [4] introduced the binary prefixes. These are based on the power of two, so that it fits with binary number representation. Each prefix is 2^{10} (1024) raised to the power of 1, 2, 3 and so on. These were introduced to give an accurate way to represent large quantities of bytes, as these are normally in powers of 2. Even today there is still confusion about using normal SI prefixes for bytes. A case of this is with data storage. Disks are sold under the SI prefixes, and the operating systems also use the SI prefixes on the surface, but actually represent values calculated using the binary prefixes. This causes a lot of confusion with consumers and gives us an incentive as developers to use the correct prefixes to avoid such confusions.

Factor	Name	Symbol	Multiplying Factor
2^{10}	kibi	Ki	1 024
2^{20}	mebi	Mi	1 048 576
2^{30}	gibi	Gi	1 073 741 824
2^{40}	tebi	Ti	1 099 511 627 776
2^{50}	pebi	Pi	1 125 899 906 842 624
2^{60}	exbi	Ei	1 152 921 504 606 846 976
2^{70}	zebi	Zi	1 180 591 620 717 411 303 424
2^{80}	yobi	Yi	1 208 925 819 614 629 174 706 176

Figure 2: Byte prefixes

7.3 QUDT

QUDT or “Quantity Unit Dimension and Type” is a model designed by the QUDT organization. QUDT was originally developed for a NASA project and is now available to the public. The model is constructed from a series of ontologies, though the two main ontologies are the following:

7.3.1 Units

The unit ontology describes the unit with its unit name, a description of the unit, a system name to ensure unique naming, making the unit addressable so to speak, a conversion multiplier and offset to bring the unit to a be an SI or SI derived unit where possible and optionally a symbol if the unit has one.
The schemas for the ontologies are written using TURLE, and for the most part follows RDF, RDFS and OWL for the notations.

7.3.2 Dimension Vectors

The dimension vectors are as described in section 7.1.1

7.3.3 Facilitate Conversion

The way that QUDT facilitates the conversion is by having “every” unit and any prefixed unit that one would like to convert stored in the models. This severely limits the usefulness of the system by increasing the workload, storage requirements and complexity of the model.

7.4 Storage

One flaw that we found in QUDT was the ever-increasing storage requirement for increasing the number of units that could be converted. To alleviate this issue, we thought to go with a more programmatic approach of parsing units.
This act of parsing units as opposed to storing a copy of every compound unit and any unit combined with all prefixes means that the system would only be required to store the base units thereby limiting the workload of entering custom units into the system as well as increasing the usefulness by allowing the conversion of compound units not initially thought of by the system creators.

8 Requirements

This section details both the functional and non-functional requirements of the system. Requirements have been decided upon through a dialogue with the supervisor that aimed to both define a valuable system and enable gradual improvement to discover how best to optimise performance of the system. From this dialogue, a use case diagram has been created to better visualise the requirements. The use case diagram provided the foundation for further use case specifications to ensure that features could be developed from the requirements. The individual use case specifications can be seen in the appendix, section 17.2 Use case specification.

8.1 Use case diagram

The use case diagram has two actors: Data Provider and Subscriber. It is important to note that these actors do not necessarily indicate human users. They would most likely be other programs or even entire systems.
The system is supposed to be able to be integrated into other system as a link in a stream of data. The two actors can therefore also be seen as input and output of the system.
The data provider feeds the system with data, which can be subscribed to. It could for example be a temperature sensor that periodically reads the temperature of a room.
The subscriber wants the data from the data provider, but it potentially has to be in a different unit. The previously mentioned temperature sensor might provide the data in degrees Celsius, but a researcher might need it in Kelvin for it to fit in his data analysis tools.

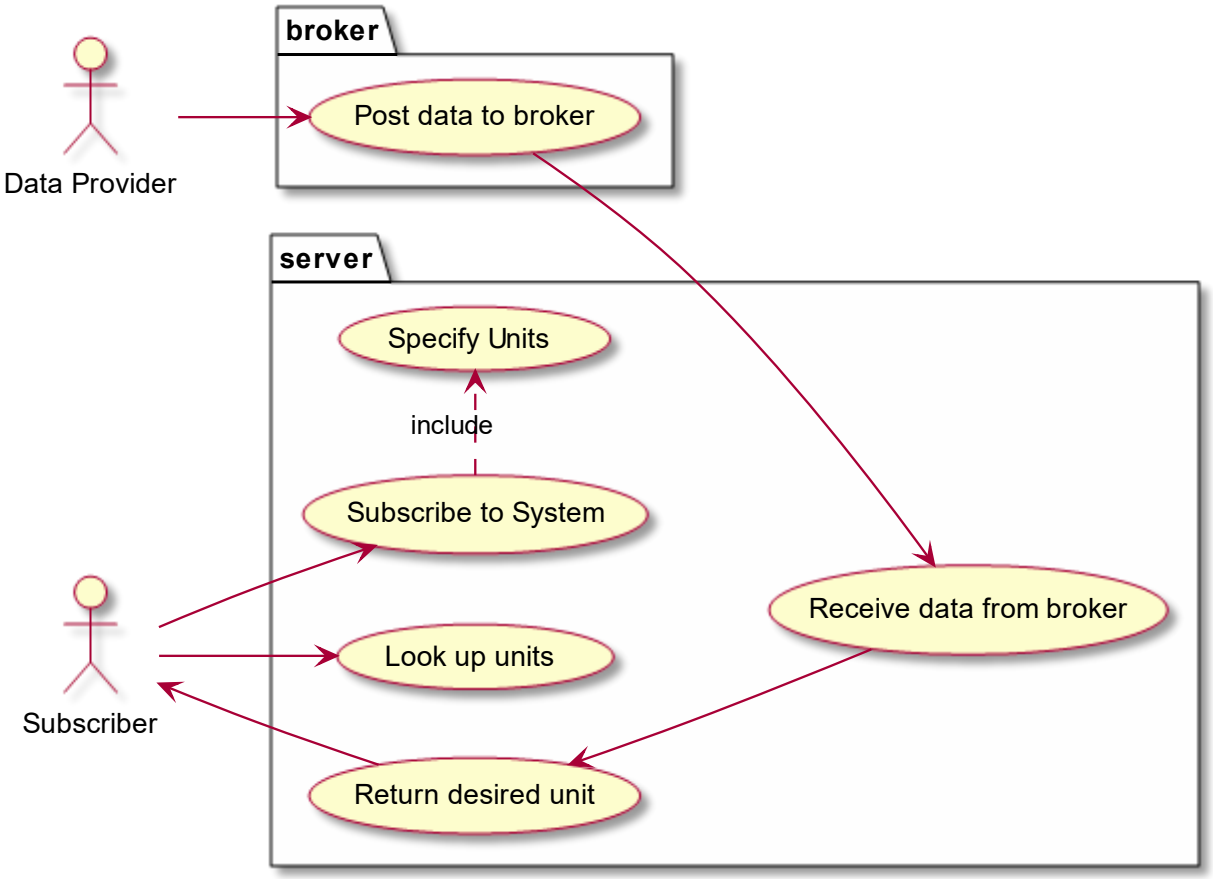


Figure 3: Usecase Diagram

The use case also describes the separation of the system into a broker and a server. The broker is there solely to post data to, so the server has a limited number of places to read data from. This part is important as there might be a large amount of data providers that otherwise would have to connect directly to the server, which might unnecessarily complicate things.

The use cases have been further specified and can be seen in 17.2 Use case specification. The standard use case goes like this:

1. A subscriber wishes to read some data in a certain unit.
2. The subscriber tries to look up the system name of the unit.
3. The subscriber subscribes to some data in the specific unit.
4. The data provider posts data to the broker.
5. The system reads the data from the broker.
6. The data gets converted to the desired unit.
7. The data gets send to the subscriber.

8.2 Functional requirements

The system is quite simple. It aims to convert any unit to any other compatible unit as an intermediary in a larger system. To be able to do that, the following functional requirements have been devised.

8.2.1 Subscribe to measurement

It should be possible to subscribe to any measurement stream with a predefined unit and have it return the values in any given compatible unit. The initial data's unit should be specified in a topic or equivalent, so that the system knows about it when the subscription occurs. The system should thereby be able to determine if the requested unit is compatible with the predefined unit.

8.2.2 Read from broker

The system should be able to subscribe to an arbitrary data stream in a system like MQTT and receive measurements, if requested to do so.

8.2.3 Prefixes

The system should be able to interpret any SI unit prefix [5] as well as any binary prefix [6]

8.2.4 Convert units

The system should be able to convert a measurement between compatible units specified by the user of the system.

8.2.5 Compound units

The system should be able to interpret compound units and thereby convert from any combination of units to any other equivalent combination of units.

8.3 Non-functional requirements

While the system only has a few concrete functional requirements, there are quite some critical demands to how the system runs. The system is supposed to work in an IoT environment, which comes with quite some non-functional requirements of the system. It needs to be fast while maintaining many different connections with other systems.

8.3.1 High throughput

The system is created with having the goal of being used in an IoT environment. It is therefore important that the system has a high throughput to ensure that it does not slow the surrounding system. It is also conceivable that the system can be incorporated in a factory line or similar, where a delay could cause loss of goods. One of the goals of this project is to be able to manage as high a throughput as possible.

8.3.2 High availability and reliability

Being part of an IoT environment also means that it is important that the system is both available as much as possible and reliable when used. The system should aim to always be available for clients. It is, of course, impossible to get 100% uptime, but the system should be implemented with the aim of it always being available.
The system should not send an incorrect value. Incorrect values are purely from a mathematical perspective. The system does not determine whether a value is physically possible. A value of negative kelvin is not considered an incorrect value if the input value really converted to it. It is better for the system to throw an error or break than to send an incorrect value. Incorrect values might cause damage, if other critical systems rely on the values to be correct. An error, on the other hand, should ideally cause a shutdown, which causes delays, but hopefully no damage.

8.3.3 Adaptable

The system should be able to operate together with many other systems. Being able to configure the system to different ip-addresses or data formats is central. It might even be useful to be able to listen to different brokers and deliver per request specific data formats. Similarly, having the system be able to deliver the data in different ways, such as through a websocket or to another broker, could come in handy.

8.3.4 High scalability

IoT systems are getting bigger and bigger. The system should support this and be able to listen to millions of sensors and deliver the same number of converted values.

8.3.5 Testable

A central part of the project is to improve performance of the system. To improve performance, it is important to be able to measure performance. Otherwise, it is difficult to know whether a change caused any improvement. The system should, therefore, be built with easy testing in mind.

9 Tools

9.1 ASP .Net Core

ASP .NET Core is an open-source cross-platform developer platform, which runs on Windows, Linux, macOS, and has official support for Docker. .NET comes with many different packages and application frameworks for all kinds of software.

Specifically, it comes with scaffolding for a web API, which makes it fast and easy to implement an API endpoint. Many of the applications, which are created using .NET, follow the design principals behind dependency injection, web APIs included. Classes and interfaces can be declared as services in the start-up of the program. They can have different scopes, which determines how long they are kept alive. The framework will then automatically inject the class(es) into any other class which has them declared in its constructor. This means that classes do not need to rely on specific implementations of classes. They just specify an interface. Any implementation of this interface will then work. It is therefore super easy to change the specific implementation. For example, if you want to use a different type of persistent storage or switch the broker system. One can simply make a new class which implements the desired interface. There is no need to change anything in any of the classes, which use the changed class, as they only depend on the interface. The .NET system then makes sure to hand the right instance of the class to other class at runtime.

9.1.1 Tasks

.NET's tasks represent a piece of work that needs to be done. It can then be started, and the work will be done. One can then either wait for the task to finish or continue execution on something else. In many ways, it is similar to a thread, but it is at a higher level of abstraction, which makes it easier to implement. It also has the benefit of coming with many additional features compared to a thread and the .NET framework has been optimized to execute tasks as fast as possible. The developer therefore does not need to worry about optimizing threads or implementing a way to get a result from the work done in a thread.

9.1.2 SignalR

SignalR is a library for .NET, which greatly simplifies the use of websockets. It makes it possible for clients and servers to call methods for each other. There is therefore no need to develop a communication standard, as is the case for a traditional websocket. SignalR has a default limit of 5000 concurrent requests per application [7].

9.2 MQTT

MQTT or "Message Queuing Telemetry Transport" is a system for software clients to publish and consume data through a central broker. The data on a broker is addressed by topics. Topics are represented by a text string which can be segmented by forward slashes to give the sense of structure like the Unix folder structure.
Data is published to the broker by a client. When the client publishes data to the broker the client specifies if the topic and if the latest message should be retained on the broker such that any client trying to access the topic will find a message instead of just waiting for new data to be published.
Data is retrieved from the broker by a client by subscribing to the data on a specific topic. If the last message on the topic was set to retain, the client will immediately receive said message and will later receive any new update to the topic.

9.3 PostgreSQL

PostgreSQL is a relational database management system. It has full SQL compliance. Postgres is an obvious choice for database management system, as it is both free and open-source. Finally, both group members have experience with Postgres, which further supports the choice.

10 Design

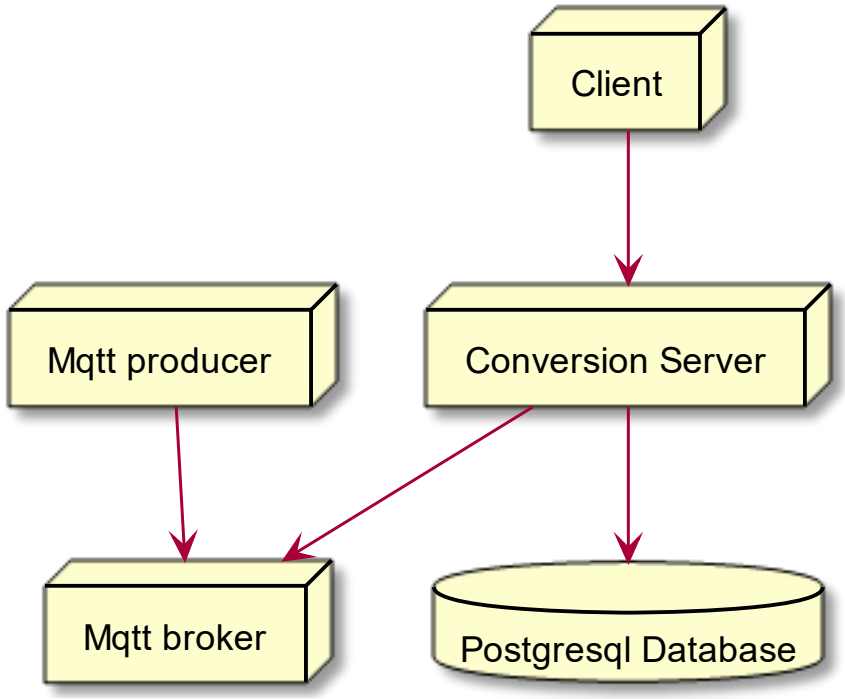


Figure 4: System overview

The system is designed around three elements. The main element is the conversion server. This is where all the logic occurs and the link between the other elements. The conversion server is connected to a Postgresql database, which contains data concerning all the available units. An SQL database was chosen because it provides a central point of truth. If more than one conversion server needs to be deployed, they could all access the same database. It is therefore only necessary to add new units in one place for it to take effect everywhere. There are several good reasons to need more than one conversion server. There might be a need for extra reliability, so a redundant server is deployed in case the first one fails. Maybe the data is sensitive, and the servers must be on separate networks.

The last central element is the MQTT broker. The important part is not MQTT. It could be any pub sub system. The conversion server just needs somewhere to read the data from without it being directly fed the data. Directly feeding the data to the system would put an unnecessary strain on the server and would require extra implementation, which is already covered by MQTT.

The last two elements in Figure 4, client and MQTT producer, are users of the system and not technically part of the system. They have been implemented as examples and to be able to test the system. The MQTT producer represents some kind of data producer, it could be a temperature sensor

for example. It will publish every new reading to the MQTT broker. The client wants to read this data in a unit, which it is not originally published in. It connects to the conversion server through a websocket and gets fed the data in real time and in the desired unit.

10.1 Websockets instead of republishing to the broker

We made the choice of going with websockets over republishing to the broker mainly due to two issues.
The first thing that made us choose to use websockets over reposting converted values to the broker was the process of subscribing to data. If the system were designed to repost data, a subscription to the original data would have to happen somehow and would probably be initiated by a call to a REST API, so that the system knows which data to convert from and to. This would have a likely consequence of wasting compute resources, as the server would possibly be converting for a client that isn't listening. With our solution of using websockets, the subscription to data ends together with the closing of the connection, thereby never leaving an open connection hanging wasting resources.
The second deciding factor would be that reposting data to the broker would require write permissions on the broker, thereby limiting the usefulness of the system. It would then only be useful in situation where one has read access to the data broker, which might not always be the case.

10.2 MQTT as starting point

Going with MQTT as a starting point was a choice of simplicity. What we needed was an existing solution for a pub sub service that was easy to integrate with, and MQTT was exactly that. Integrating MQTT into our system took little effort and offered us the ability to receive messages without much overhead, and it also allowed us to quickly create a data providing client in Python that could serve as a simple data source that would be easy to tune as the tests went on. We chose to not go with the other solution that we knew of i.e., Kafka [8]. As Kafka is a system of much greater complexity and a different set of features, which far exceeded the needs we had. Implementing Kafka would have required us to make decisions about how to support the more complex features and would thus have taken away from the time spent on the more core problems.

10.3 QUDT as data system

Using QUDT as a starting point was very advantageous to our project. The structure of the values and the way QUDT went about facilitating their conversions was very enlightening and gave us an understanding of how conversions are commonly done, as we later found other solutions such as WolframAlpha using a similar system of conversion factors and dimension vectors. With that being said QUDT also had some shortcomings, as the system was designed to contain the specific units you would want to convert it required an ever-growing list of units to accomplish a goal of universal unit conversion. The need to have every permutation of prefixes and different unit systems registered for every conceivable compound unit renders the system rather unusable for conversion of anything more than commonly used units.

10.4 Class diagram

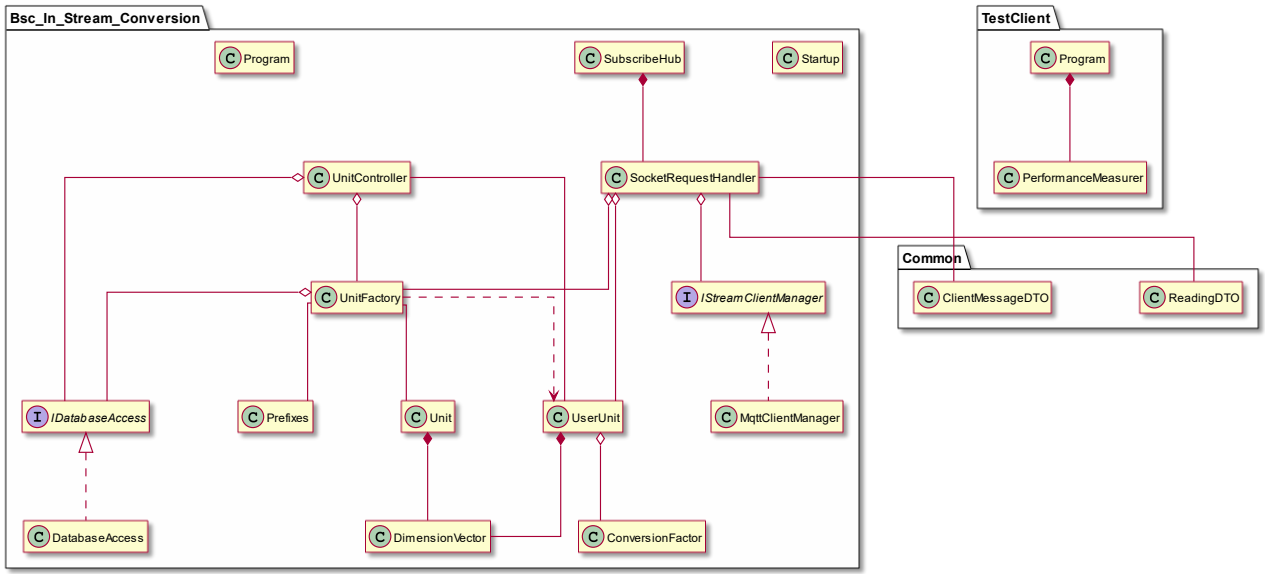


Figure 5: No class details & Ignoring dependency injection.

Figure 5 shows the complete class diagram of the system except for dependency injection relations. Dependency injection is handled by Startup, which therefore has relations to many classes that do not give an accurate representation of the system. From this diagram, one can see how everything is initiated by either UnitController or SubscribeHub. These are the two ways other programs can interact with the program. They both eventually use the UnitFactory to create a UserUnit as the means to convert whatever requested unit.

The common package is for data transfer objects (DTO), which are used when data is shared across programs. ReadingDTO represents the data that is sent to the broker by for example a sensor. ClientMessageDTO is the object sent from the web API to subscribing clients whenever it has converted the data from a ReadingDTO.

10.5 Unit representation/notation

For the computer to be able to work with the units, the user needs some form of structured notation to describe a unit. This notation should be both easy for a human and the computer to understand. If it is difficult for a human to understand, the system becomes a bother to use, and the likelihood of mistakes increase. If it is difficult for the computer to parse, it becomes difficult to implement for the developers and it might require more processing power.

Luckily, units are relatively simple and easy to represent. All combined units can be represented as a single fraction, which means that a single '/' can be used to separate the numerator from the denominator. Each unit in the numerator or denominator can then be separated by an '*'. The unit will then look like a normal mathematical expression, which a human can easily read, while still being simple to parse for the computer. To make it even easier for the human, powers can be represented by a '^'. One does then not have to write the same units multiple times. Meters per second squared would then look like "M/S^2" and joule per kelvin per kilogram would look like "J/K*KG". It is worth noting that each unit known by the system is represented by one or more letters. These are known as the unit's "system name". System names are all capital letters, which means that any unit entered by the user is case insensitive.

This representation is completely arbitrary. A couple of factors influenced it to be what it is. The mathematical symbols were chosen as these are the most commonly used and would therefore make it easiest for a human to read. A notable omission is that the notation does not include parentheses. These were excluded to make it easier for the computer to parse, but it comes with the downside that "J/K*KG" could easily be mistaken for "(J*KG)/K" even though it means "J/(K*KG)" in the system. The decision to make all system names capital letters was taken by QUDT. There seemed to be little reason to go over all the units and give them different names, even though it might shorten some of the system names, as each letter could be used twice.

10.6 Database

The PostgreSQL database is used for storing the units and their conversions. The database schema is made to closely match QUDT, as all the data comes from QUDT. This means that the database contains three tables: Units, DimensionVectors, and QuantityKinds. The unit table contains the conversion rate and offset. It also includes a small description of the unit. The dimension vector has a column for each dimension, as described in Dimension vector. It also has a foreign key tying it to a unit in the unit table. Quantity kinds are a form of category of what the unit measures. They are closely tied to dimension vectors but are more humanly readable. Some units also fit more than one quantity kind and quantity kinds can subcategories of another quantity kind. For example, altitude is a subcategory of length, and both can be measured in meters.

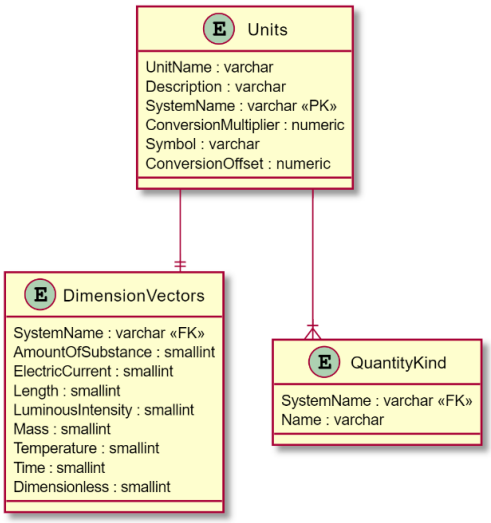


Figure 6: EER diagram over the database

10.7 The unit classes

The system is naturally designed around a Unit class. The unit is so important that the system even has two different unit classes. The class just called "Unit" is basically a database entity, which reflects how units are stored in the database. Objects of the Unit-class are only used as a go between SQL queries and the UserUnit-class to make the code easier to read and more reusable. Figure 7 shows a UML representation of the "Unit" class. UnitName is a written-out name of the unit meant to be humanly readable. Symbol and SystemName are similar, but SystemName is the name described in 10.5 Unit representation/notation and is meant to be easy for the computer to use, where symbol is meant to be the common shorthand used by humans. Description is a short text which describes the unit. The rest of the attributes are direct mappings of concepts laid out in the Domain analysis section.

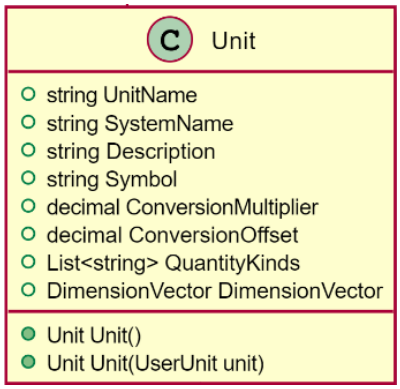


Figure 7: The Unit Class

The class called “UserUnit” is where the magic happens. This class represents any combination of units and can be multiplied or divided with another instance of the class to create a new unit. It also handles prefixes for both the numerator and denominator. Finally, it has a multiplier and an offset. This makes the class able to convert both from and to the base unit through the methods ConvertTo-BaseValue() and ConvertFromBaseValue(). One can thereby easily combine the desired units and then be able to convert back and forth. The “UserUnit” class can be seen in Figure 8.

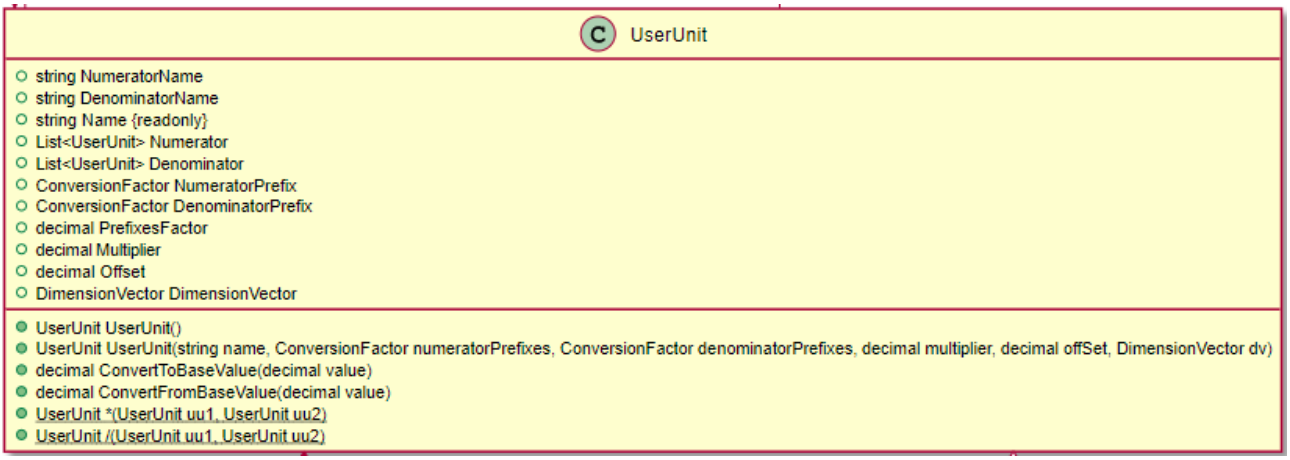


Figure 8: The UserUnit class

10.8 Prefix conversion factor

The unit classes alone would become too complicated to be nice to work with if they also had to include prefixes. The prefix conversion factor class was therefore introduced. If prefixes always were a factor of the same power, this class would not be necessary as one would then be able to represent them as a single integer. However, prefixes are different when you work with bytes compared to all other units as described in 7.2.1 Prefixes. The ConversionFactor class abstracts this away. It enables the program to combine factors with different bases, which is necessary to offer the freedom of combining any two units including their prefixes.

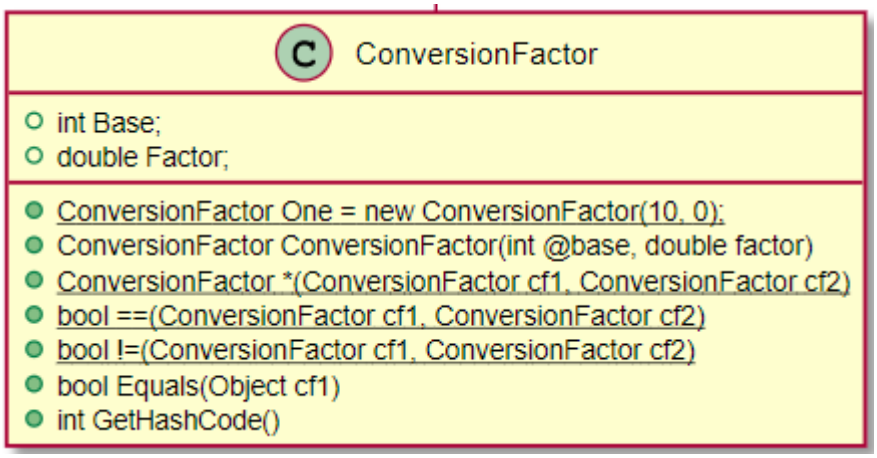


Figure 9: The ConversionFactor class

10.9 Unit flow

When the system has to parse a string representation of a unit, the first step is to separate base units. It then looks up the base units one at a time in the database. The database accessor returns a Unit object. The attributes of the unit object are then passed to a new UserUnit object along with a prefix if a prefix was part of the unit. An example of this can be seen in Figure 10. This example uses kilojoule. The example omits the UserUnit's Name and PrefixesFactor attributes, as they are both derived from other attributes.

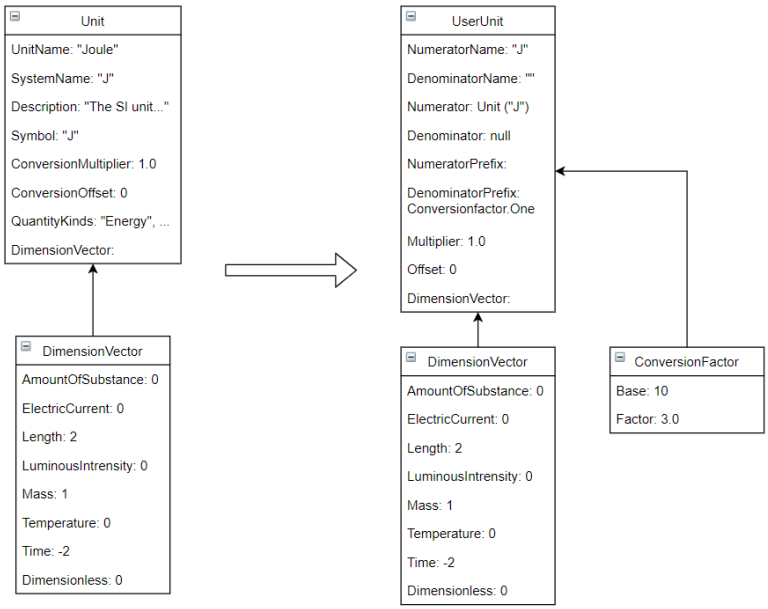


Figure 10: Unit to UserUnit

UserUnit objects can then be multiplied or divided together according to the string representation of the given unit. These two operations are similar, but inherently opposites. When two UserUnits are multiplied, the numerator names are appended together, and the denominator names are appended together. When they are divided, the numerator of the first UserUnit and the denominator of the second UserUnit is appended together, and vice versa. The same happens with the prefixes, except they are multiplied together instead of appended, as they are numbers. The multiplier is either multiplied or divided in accordance with the operation. Offsets are ignored as soon as two UserUnits are combined, as it then becomes a ratio. When it is a ratio, the absolute value does not matter anymore, only the difference between values. The offset does not have an influence on the difference, so it can be ignored. DimensionVectors are added together when multiplied and subtracted when divided. This is because they are technically powers, and basic mathematics tells us that:

$$a^x \times a^y = a^{x+y}$$

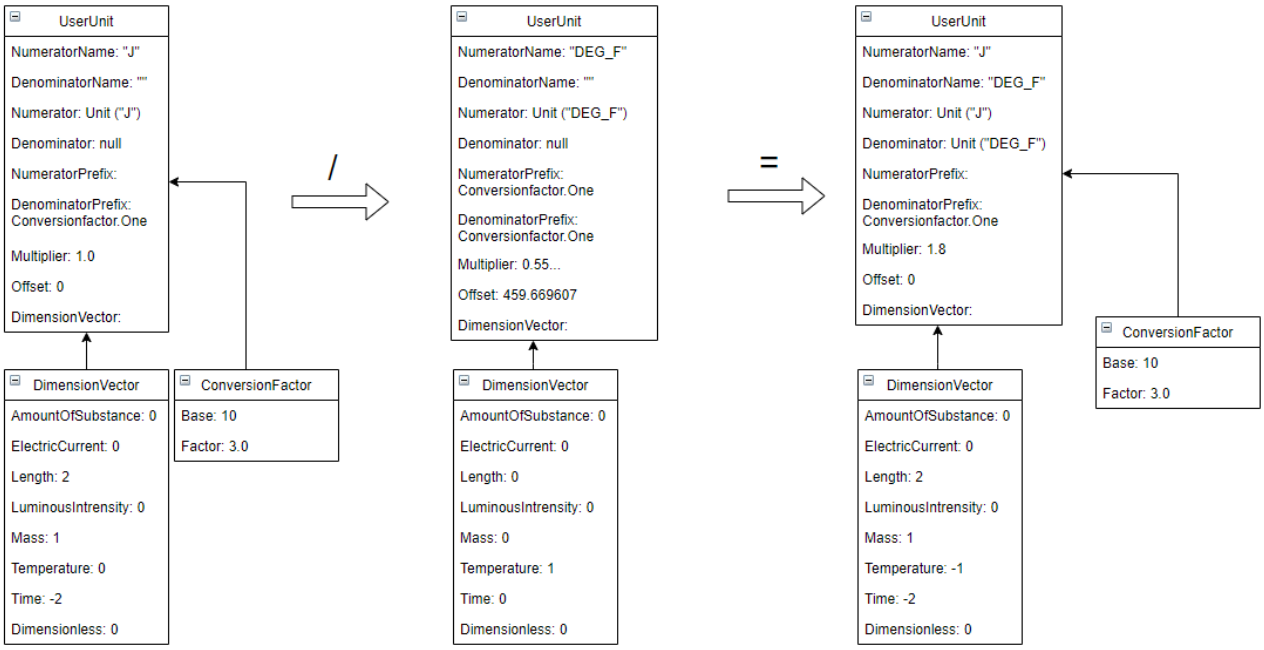


Figure 11: Dividing UserUnits

Continuing the kilojoule example, Figure 11 shows a UserUnit representing kilojoule being divided by a UserUnit representing degrees Fahrenheit.
The kJ/°F UserUnit can then be multiplied or divided by another unit and another unit until the desired combined unit has been achieved. In this way, it is possible to represent any unit, which consists of the base units already in the system.

10.10 Dependency injection

A central part of .NET is its encouragement of dependency injection. Dependency injection is when the implementation of a class is abstracted away behind an interface. This makes classes less dependent on each other, as they are only dependent on an interface, which makes it easier to change or switch out classes. The actual implementation is then injected into a given class at runtime by an injector. This means that an instance of a class implementing a desired interface is instantiated by .NET and passed through the constructor to any other class which has the interface as a parameter in its constructor.

10.10.1 Registering services

.NET refers to classes which can be dependency injected as services, and they are stored in a ServiceCollection. A service is .NET's term for a class, which is injected into other classes through dependency injection. The ServiceCollection interface is the interface developers use to register classes as services, so that .NET can inject them into classes that need them. Services are registered in the Startup class in the method ConfigureServices(). .NET will then be able to look up whether it has an implementation of an interface, whenever a class has one or more interfaces as parameters in its constructor. The class will then have the services passed through its constructor, as it is being instantiated. When one registers services in .NET, one can choose one of three lifetimes for the service. A lifetime determines when a service is instantiated and how long it is going to be reused. They are: Transient, Scoped, and Singleton.

- Transient services are created each time that they are requested. Every time a class wants an instance of a transient service, a new instance is created.
- Scoped services are created once per client request. For example, in a web API, a scoped service lives from it is requested until the API has returned an answer to the client. This makes it easy to separate different requests, so that they do not interfere with each other.

Singleton services are created once, which means the same instance is used every time the service is requested. Singletons are good for accessing limited resources, such as a database, where there is a chance of sending too many requests at once if the program accessed it at will.Figure 12 shows the

```
public void ConfigureServices(IServiceCollection services)
        {

            services.AddControllers();
            services.AddSignalR();
            services.AddSwaggerGen(c =>
            {
                c.SwaggerDoc("v1", new OpenApiInfo { Title = "Bsc_In_Stream_Conversion", Version = "v1" });
            });
            services.AddSingleton<IStreamClientManager, MQTTClientManager>();
            services.AddSingleton<IDatabaseAccess, DatabaseAccess>();
            services.AddSingleton<UnitFactory>();

            services.AddScoped<SocketRequestHandler>();
            Log.Logger = new LoggerConfiguration()
                      .WriteTo.File(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +  $"/logs/log-.txt",
                      rollingInterval: RollingInterval.Day)
                .CreateLogger();
        }
```

Figure 12: ConfigureServices()

ConfigureServices() method. Services are easily registered by providing both the interface and the implementation class. Lines 37-42 registers built-in .NET services for our web API and SignalR websocket. Lines 43-47 registers service which we have implemented. As can be seen, it is also possible to register a class as a service without an interface. Classes can then reference the class directly and get it injected without using an interface. This can be beneficial for prototypes where an exact interface might not have been determined yet. It can also be used if one is very sure that it does not need to be switched out, though that is rarely possible as things will often change.

10.10.2Requesting services

Requesting a service is as easy as putting the interface in the constructor. Then .NET takes care of the rest and injects the service at runtime. This can be seen in Figure 13: Injecting Services.

```
public SocketRequestHandler(IStreamClientManager MQTTClientManager, UnitFactory unitFactory)
        {
            this.MQTTClientManager = MQTTClientManager;
            this.unitFactory = unitFactory;
        }
```

Figure 13: Injecting Services

This constructor is never directly called in the solution. It is only ever being called by the .NET framework. .NET will look for an implementation of IStreamClientManger and the UnitFactory-class in its ServiceCollection. It will then, if necessary, instantiate objects of these classes and pass them to the SocketRequestHandler. As both services are registered as singletons, they might already have been instantiated and can thus be passed directly.

10.11 Receiving requests

The system is designed to have two entry points for other programs: UnitController and SubscribeHub. UnitController is a standard API-controller for http requests. This class is intended for one time conversion lookups and for querying the information needed to do said lookups. SubscribeHub is a SignalR-hub which is used for connecting to a topic on a broker and continuously receiving data in the specified unit. The SubscribeHub then delegates a unique instance of the SocketRequestHandler-class to each connection. A SocketRequestHandler is responsible for initializing everything needed to convert and deliver the data. Both the SocketRequestHandler and the UnitController uses the UnitFactory class to build a UserUnit object from the user's input. The UnitFactory both retrieves the data stored in the database for each unit and combines them together into a single unit. The SocketRequestHandler and the UnitController is thereby not concerned with the units, only that they provide a method to convert to and from then unit. After having initialized the to and from units, the SocketRequestHandler subscribes to the desired topic through the IStreamClientManager interface. This interface makes it easy for the system to include support for different network protocols. This sequence is described in Figure 14.

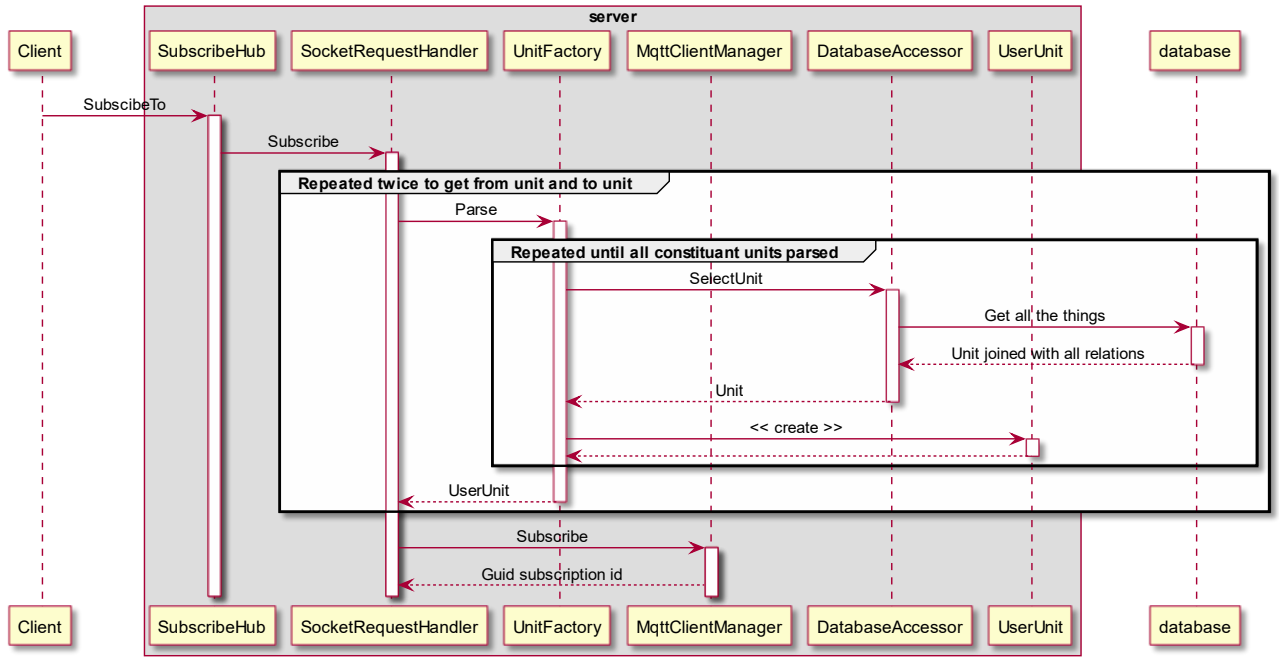


Figure 14: Subscription sequence diagram

10.12 Message Handling Sequence

When the system has had requests for subscriptions, it starts listening on the broker for new data. When it receives new data, it finds all the clients subscribed to said topic and starts converting the data into the unit, which each individual unit has requested. As soon as the data has been converted, it is sent to the client.

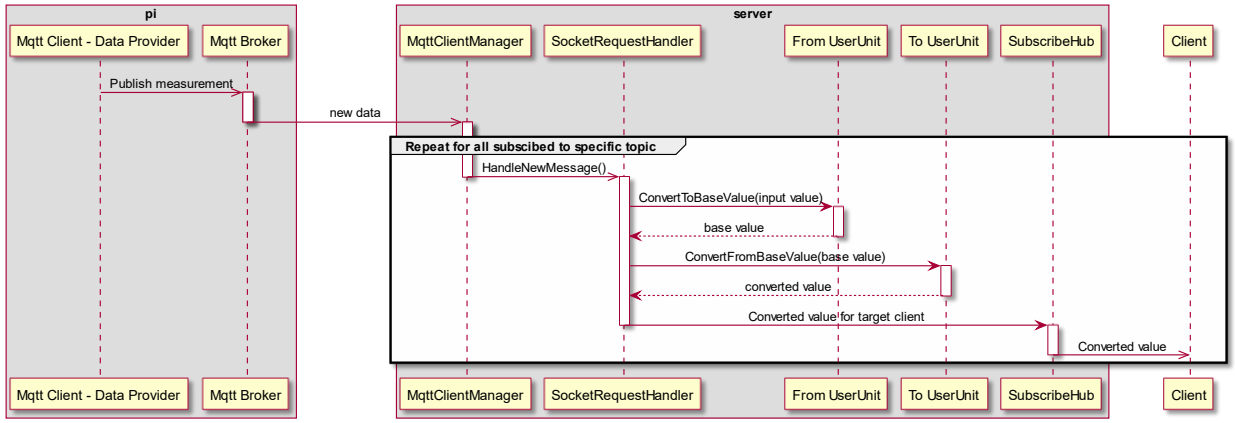


Figure 15: Message Handling sequence diagram

11 Implementation

In the following section, the implementation of essential parts in our solution will be described, and the choices regarding the implementation will be clarified. The thoughts behind the implementation will likewise be presented alongside concrete examples.
The software was developed in three iterations, here after referred to as prototypes.

11.1 Subscribing

The main logic behind subscribing is in the SocketRequestHandler class which was initially introduced in the first prototype. SocketRequestHandler is registered as a scoped service in .NET. Registering a service in this way means that each client has its own instance of the class. When a new client connects through the SubscribeHub, it gets a new instance of the SocketRequestHandler class. The Subscribe() method is then called immediately afterwards.

```
public async Task Subscribe(string topic, string toUnit, Func<string, object[], CancellationToken, Task> answerCallback)
{
    this.topic = topic;
    this.toUnit = await unitFactory.Parse(toUnit);
    this.answerCallback = answerCallback;
    FromUnit = await unitFactory.Parse(topic.Split("/").Last().Replace("%F2", "/"));
    if (this.toUnit.DimensionVector != FromUnit.DimensionVector) throw new InvalidOperationException("Units do not have the same dimension vector");
    subscribtionId = await MQTTClientManager.Subscribe(topic, HandleNewMessage);
}
```

Figure 16: Subscribe method - third prototype.

The Subscribe() method takes the topic, the unit to be converted to as a string, and a function as parameters. The function is the method which sends a response to the client. It is passed Clients.Caller.SendCoreAsync, which is a method that is provided by SignalR to send data back to the client. In the second version the UnitFactory was introduced and is here used to parse both the unit, which is converted to, and the unit, which is converted from. Prior to the introduction of the UnitFactory the UserUnit was responsible for doing a shallow parse of the from unit only.
The unit which the data is originally in can be determined from the topic. This extraction from the topic is possible because the topic is designed such that the last level is the system name of the from unit. This is done to alleviate the need for the client to know what specific unit a temperature sensor for example. The UnitFactory (or for the first prototype UserUnit) then for the last two prototypes turns the string representations into UserUnit objects and fetches the conversion rates from the database. The objects can then easily be used to convert the units of the data as soon as any data is posted to the topic. In the third prototype a check for the dimension vector was also introduced to ensure unit compatibility. Otherwise, an exception will be thrown, and an error message will be sent back to the client.
Finally, the method uses the MQTTClientManager to subscribe to the topic, so that it can start receiving data. All of this is done by the 6 lines of code shown in Figure 16.

11.2 New data posted to the broker

The MQTTClientManager subscribes to the MQTT broker, as soon as the system receives its first subscription, as describe in the previous section. This means that the MQTTClientManager has an event that fires every time data is posted to any topic. This event has been overridden to look at MQTTClientManager list of subscribed SocketRequesthandlers and calls the HandleNewMessage() method on all the ones which are subscribed to the topic which caused the event to be fired.
The HandleNewMessage() method is asynchronous, but it is not awaited when called. .NET will therefore keep starting new tasks for each call to the method. It is also responsible for deciding when each task is allocated processing time.

The method HandleNewMessage() is one of the software units that evolved the most throughout the three prototypes, and for the final testing, the older methods were altered to support the new testclient, to ensure consistent measurements.

11.2.1 Prototype 1

```
private async Task HandleNewMessage(string message)
{
    try
    {
        var msgInObj = JsonConvert.DeserializeObject<ReadingDto>(message);
        var userUnit = UserUnit.Parse(toUnit);
        var value = msgInObj.Reading;

        var numeratorValue = await unitConverter.Convert(FromUnit.Numerator, userUnit.Numerator, value);
        var denominatorValue = await unitConverter.Convert(FromUnit.Denominator, userUnit.Denominator, 1);

        var fromUnitPrefixfactor = (decimal)Math.Pow(FromUnit.NumeratorPrefixes.Base, FromUnit.NumeratorPrefixes.Factor) /
                                   (decimal)Math.Pow(FromUnit.DenominatorPrefixes.Base, FromUnit.DenominatorPrefixes.Factor);

        var toUnitPrefixfactor = (decimal)Math.Pow(userUnit.DenominatorPrefixes.Base, userUnit.DenominatorPrefixes.Factor) /
                                 (decimal)Math.Pow(userUnit.NumeratorPrefixes.Base, userUnit.NumeratorPrefixes.Factor);

        var convertedValue = fromUnitPrefixfactor * toUnitPrefixfactor * numeratorValue / denominatorValue;

        var clientMessage = new ClientMessageDto(msgInObj, Thread.CurrentThread.ManagedThreadId, MQTTClientManager.GetActiveClientsCount(), convertedValue);

        await answerCallback("NewData", new object[] { JsonConvert.SerializeObject(clientMessage) }, CancellationToken.None);
    }
    catch (Exception e)
    {
        Log.Error("Error sending message " + e.StackTrace, e);
    }
}
```

Figure 17: HandleNewMessage method - First Prototype

The method starts by deserializing the received data, this was retroactively added to the first prototype as keeping the message-content and testclient consistent was important for the performance testing. The method then parses the desired unit, as the unit has so far been stored as a string. After this the method calculates the value for the denominator conversion, numerator conversion as well as prefixes for both units. These values are used to then convert the input from the message to the final value. After that, the converted value is then packed into a message object (Again retroactively introduced for compatibility) and sent to the requisite subscriber.

11.2.2 Prototype 2

```
private async Task HandleNewMessage(string message)
{
    try
    {
        ReadingDto msg = JsonConvert.DeserializeObject<ReadingDto>(message);
        var convertedValue = ToUnit.ConvertFromBaseValue(FromUnit.ConvertToBaseValue(msg.Reading));
        var clientMessage = new ClientMessageDto(msg, Thread.CurrentThread.ManagedThreadId, MQTTClientManager.GetCurrentThreadCount(), convertedValue);
        await answerCallback("NewData", new object[] { JsonConvert.SerializeObject(clientMessage) }, CancellationToken.None);
    }
    catch (Exception e)
    {
        Log.Error("Error sending message " + e.StackTrace, e);
    }
}
```

Figure 18: Second Prototype

Like with the first version the method deserializes the message. Where the improvements to the second version then come into play is with the improved conversion system. Here the method uses the FromUnit object to convert the message to the base value, and the ToUnit is used to convert the value to final converted value. The rest is the same as the first version for the same reasons.

11.2.3 Prototype 3

The third version holds no functional differences here compared to the second prototype. Though this is the prototype in which the transmission objects were introduced. These are simple data structures introduced to assist in parsing incoming data as well as to make the job easier for the testclient.

11.3 Getting data from QUDT

QUDT makes their data available through their GitHub in a TURTLE format. The system uses its own database, but that database needs to be seeded. As the system just needs the data once, a custom parsing script for the file containing units was made, instead of setting up something, which has been designed to read the TURTLE format. The script can be seen here;
https://github.com/Hounsvad/Bsc-In-Stream-Conversion/blob/master/Backend/QUDTParser.cs
The script reads the QUDT units file and stores them in the system's database. In theory, the script is very simple. Dots separate different elements, so each time a line contains a single dot it is a new element. Unit elements start with "unit:", so the script uses this to determine if the element is a unit. It then looks for the attributes which are needed by the system and saves them to the unit object, which is then later used to make SQL insert queries.
However, some irregularities had to be accounted for. Some of the conversion multipliers have more significant figures than C# can work with, so they had to be cut off. A few of the descriptions took up more than one line, which also had to be accounted for. After this automated transfer, a manual pruning had to happen as well. The QUDT dataset contains compound units, which is irrelevant for our system, and which could result in possible strange behavior. So, the database was manually pruned to remove anything that was not a base unit, except for the gram, as this is a weird naming artifact of the metric system.

11.4 Deployment Diagram

In order to run or test the system we needed to have an environment for it to run in. As such we setup a Mosquitto server for the MQTT broker and a Virtual machine for the database.

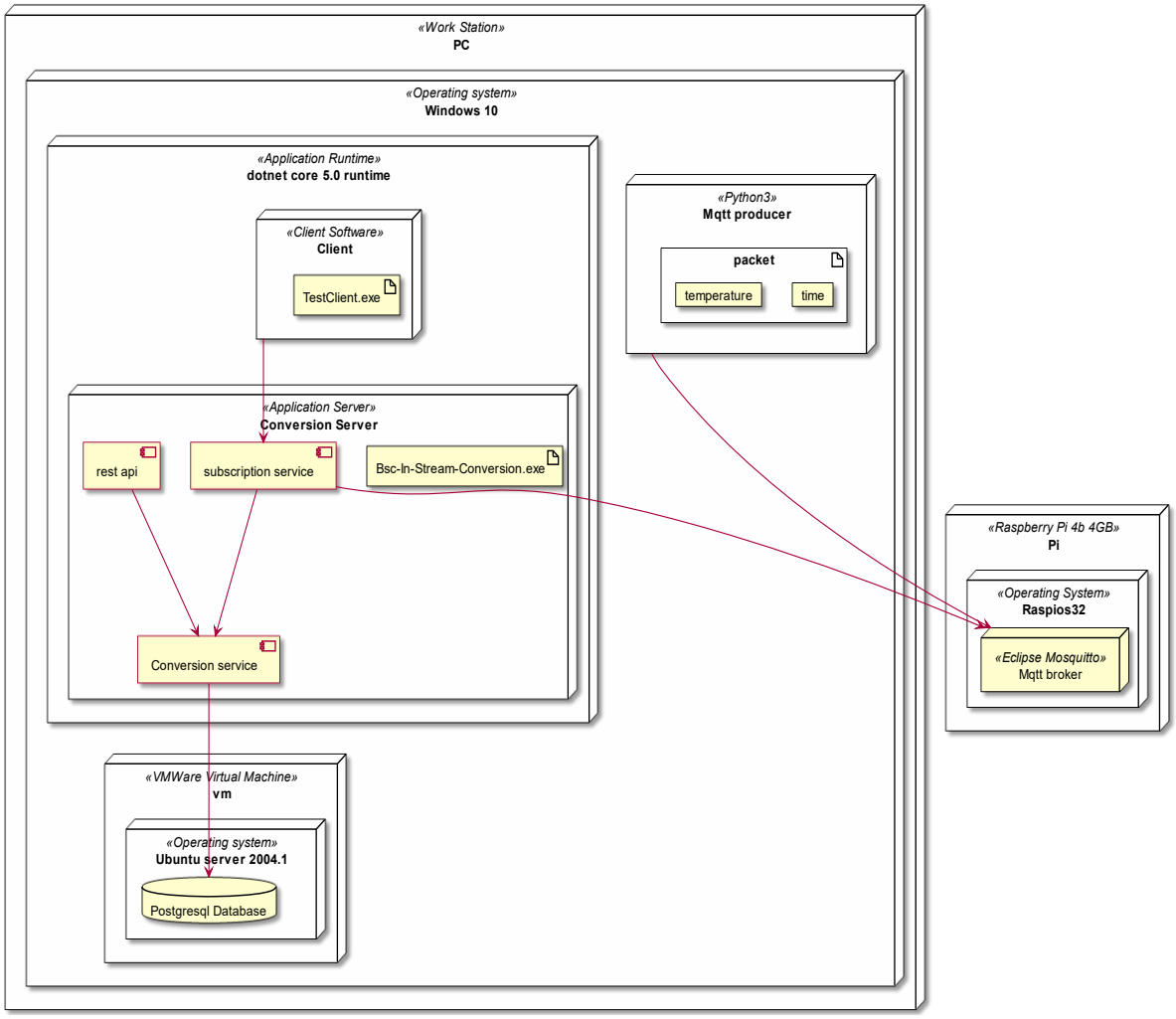


Figure 19: Complete system deployment diagram.

These were setup to fit into existing development environment and as such the Mosquitto server ended up on a raspberry pi accessible from the internet, as well as a database on the virtual machine that was cloned to a database server running the internet for non-performance testing purposes.

12 Test

In this section we aim to explain the different ways we tested our solution, both correctness and performance testing.

12.1 Performance testing

To test the performance of the system it needs to be looked at as a whole, as such the performance test is run on single computer with the exception of the MQTT broker which is hosted on a different system on the same network. The clients and server are hosted on the same computer to avoid time synchronization issues, and the broker is hosted on a different system as it is a real data source used for other things outside the project. The database is run in a virtual machine for project irrelevant setup reasons, which should emulate the database being on the same network, though this could also affect the performance of the system as it is using the same resources.

To test the system, we use a testclient this test client is a program designed to interface with our system to establish 50 subscriptions in order to gradually load the system. To stress the system, we create these test clients and let them run until they have created their connections, and then we log the message transfer times for all messages received by the client within 30 seconds are logged, after which the client then becomes “passive”, as in that it is still connected, but no-longer does anything with the received messages. Too ensure that the measurements are done at exactly the connection count stored, the system waits for the measurement interval to be done, before creating an additional testclient. this process repeats itself up to 80 different test clients which is equivalent to 4000 connections.
Due to an odd interaction of the SignalR, we experienced that the greater the number of connections a single client opens, the more likely they were to fail in connecting. Because of this odd behavior we decided to go with clients only creating 50 connections.

The MQTT client which produces the test values, is a python script which runs on the same computer as the server to ensure that they pull from the same time source so as to not have issues with time synchronization, as the timestamps are used to calculate the time it takes the messages to go from the MQTT source client to the testclient.

This stress test was performed four times per prototype version (once pr frequency) to analyze potential differences in performance depending on the frequency of the messages, as we suspect that variance in frequency of the messages might affect system performance. The test was performed with a message frequency of 1, 2, 5 and 10 Hz, where the message frequency was set by how often the MQTT source client published the messages to the broker, and the frequencies chosen were chosen as they were the factors of 10, with 10 being the highest as the data showed a fairly constant message throughput, so increasing the frequency further seemed unnecessary.

To allow for comparison we will show the runs for 5 Hz here, but all the runs can be found in the appendix at section 17.3

12.1.1 Prototype 1

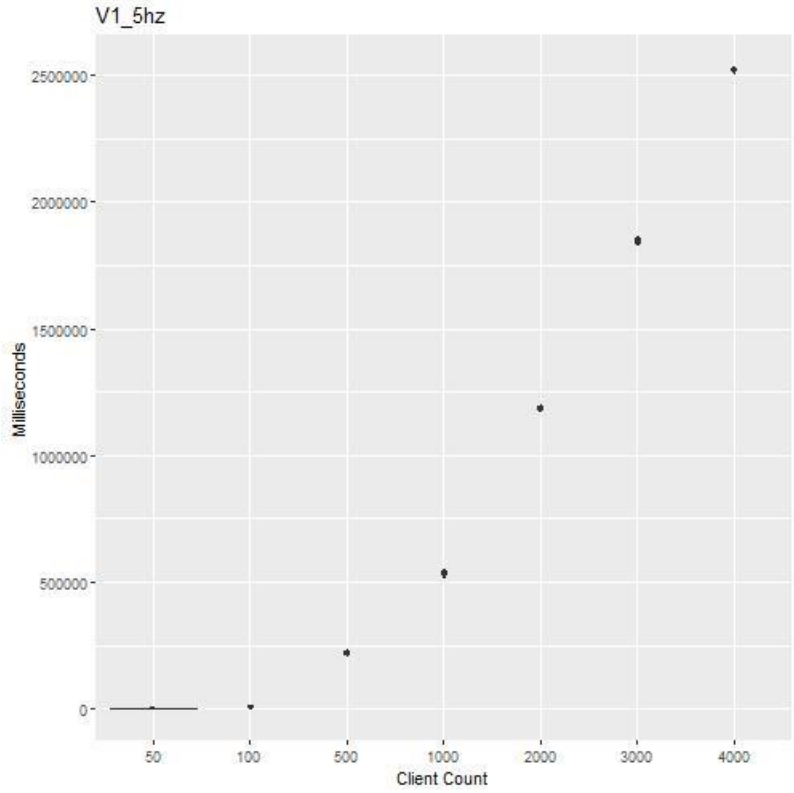


Figure 20: V1 5Hz measurement

For the first prototype the full graph is not that useful. At 5 Hz the total travel time of the message from the MQTT source client to the testclient, should be at or below a maximum of 200 milliseconds, but as can be seen on the graph, the measurements above 500 client connections are in the hundreds of thousands of milliseconds.

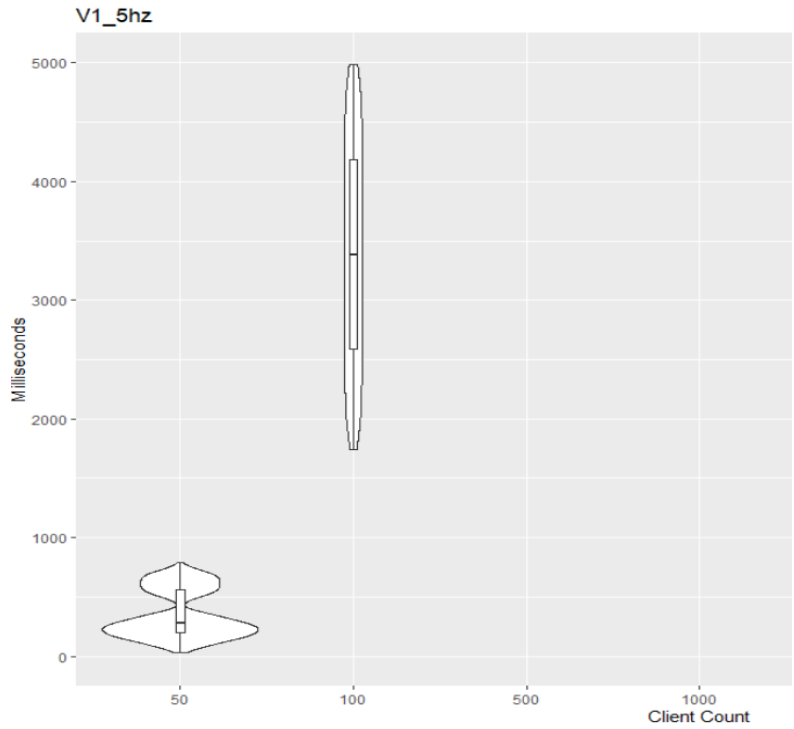


Figure 21: V1-5Hz clipped to 5000 ms

To get a better idea of the performance of the first version of the software we will refer to Figure 21 where the graph has been clipped to only include values below 5000 milliseconds. On this graph we can see that at 50 client connections we have an average of around 300 milliseconds and easily up to around 750 milliseconds showing that the first prototype was completely unusable for data rate of around 250 messages pr second.

12.1.2 Prototype 2

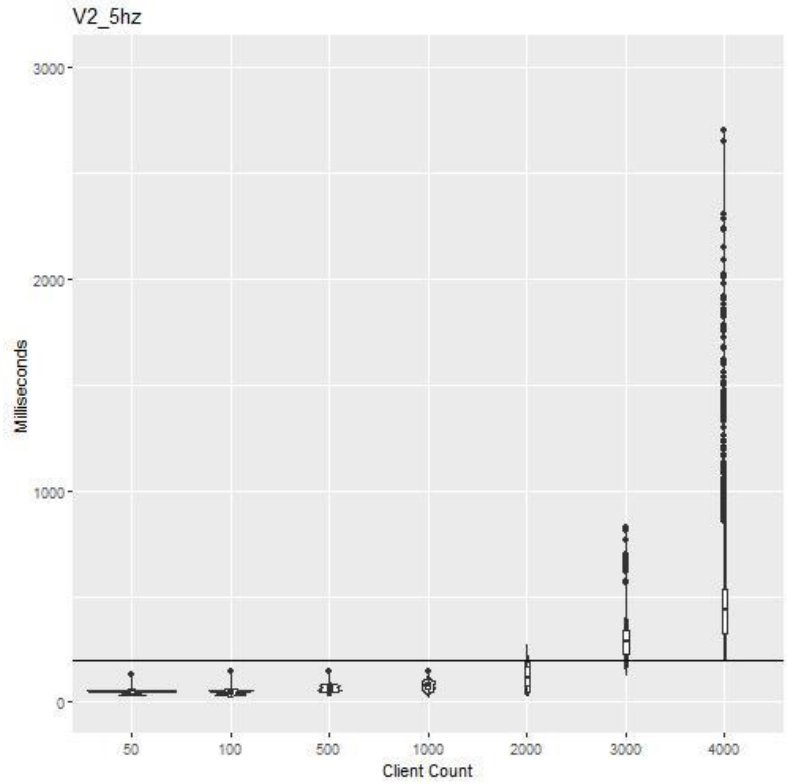


Figure 22: V2 5hz measurement

For the second prototype a graph clipped at 3000 milliseconds is significantly more useful as it encompasses almost all the outliers for the measurements. The graph with the help of the indicator line shows us that up to 1000 clients can be connected at a message rate of 5 Hz without any issues, and that up to 2000 clients can be connected with only few messages being delayed to the point of exceeding the limit of 200 milliseconds signified by the horizontal line. This tells us that the second version can handle a message rate between 5000 and 10000 messages pr second.

12.1.3 Prototype 3

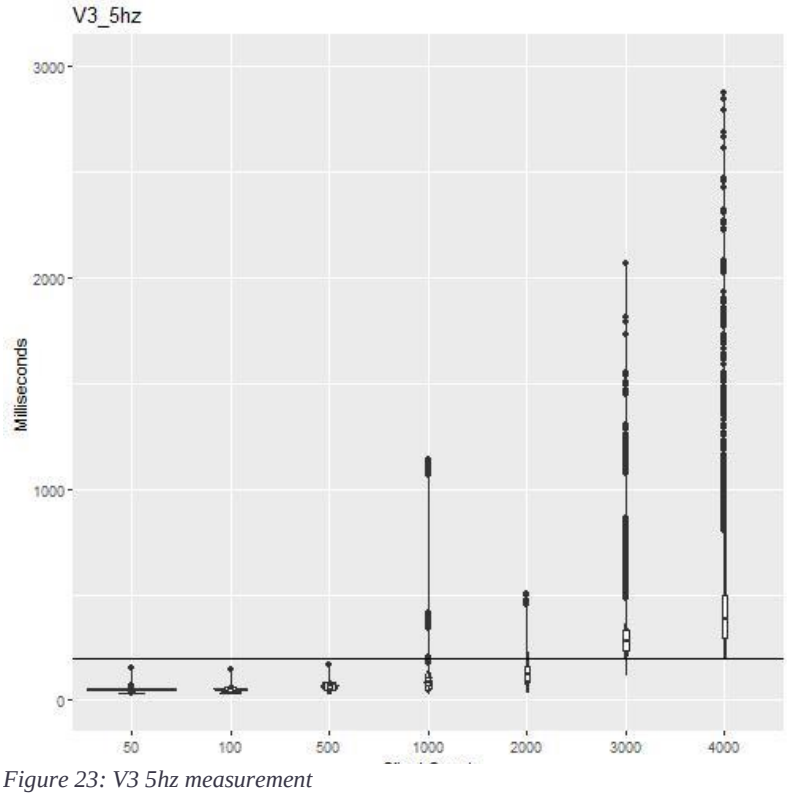


Figure 23: V3 5hz measurement

The graph for the third prototype has also been clipped at 3000 milliseconds, the same as the second prototypes graph. On this graph of the third prototype, we see that the measurements are a lot tighter within their respective groups despite the outliers. It also shows that we can just barely handle up to 2000 clients giving us a message rate of 10000 messages pr second.

12.1.4 Testing with multiple topics

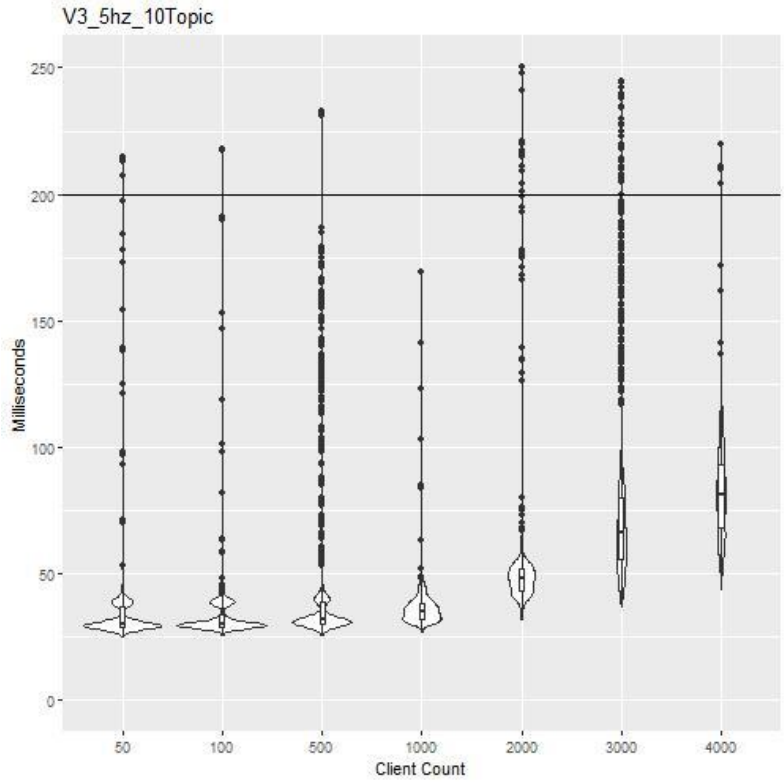


Figure 24: V3 5hz 10 Topics

As part of our testing, we also went about testing how multiple topics would affect the performance compared to our initial tests. As can be seen from Figure 24 having the testclients subscribe to multiple different topics instead of just a single topic, significantly increases the performance of the system giving us a data rate far in excess of 20´000 messages pr second. This also shows us that the system scales poorly when only single topics are request by a large number of clients.

12.2 Unit Testing

The testing strategy pursued for this system is one of tactical testing rather than complete correctness. To make sure that the improvements of each prototype did not introduce bugs, a unit test was written. The test tests the parsing of the unit and makes sure the resulting multiplier and offset is correct. There were three main reasons this method was chosen for testing.
It is essential for the rest of the system to function correctly. All the conversions are calculated based on the result of this method, so if it does not work correctly, the rest of the code cannot produce anything useful.
This method contains the most complex logic. Most of the rest of the code is either calls to libraries or basic mathematical operations. However, this method is full of conditionals, which means there is a higher likelihood of introducing bugs.

Finally, this method went through several iterations. Having a unit test made it easier to make these changes, as we could easily test if it still works.

```
public async Task CanParseInput(string input, decimal multiplier, decimal offset)
{
    IDatabaseAccess db = new MockDatabase();
    var unitFactory = new UnitFactory(db);
    PerformanceMeasurer.StartTime = input.Replace("/", "47");
    Stopwatch w = new Stopwatch();
    for (int i = 0; i < 10000; i++)
    {
        w.Restart();
        var unit = await unitFactory.Parse(input);
        w.Stop();
        PerformanceMeasurer.Log(w.ElapsedTicks, input);
    }
    var unit1 = await unitFactory.Parse(input);
    await PerformanceMeasurer.DumpLog();
    Math.Abs(unit1.Multiplier - multiplier).Should().BeLessThan(0.0000000001m);
    Math.Abs(unit1.Offset - offset).Should().BeLessThan(0.0000000001m);
}
```

Figure 25: Unit Test

Normally the UnitFactory class accesses the database, but as we do not want to rely on the database for unit tests, a mock database is passed instead. The reason that the result is checked with a less than instead of an equal is that the system uses decimal numbers, which might lose a little bit of accuracy during calculations.

```
[Theory]
[InlineData("M", 1, 0)]
[InlineData("S", 1, 0)]
[InlineData("FT", 0.3048, 0)]
[InlineData("KILOFT", 0.3048, 0)]
[InlineData("PICOFT", 0.3048, 0)]
[InlineData("FT^2", 0.09290304, 0)]
[InlineData("DECIFT^2", 0.09290304, 0)]
[InlineData("FT^3", 0.02831684659, 0)]
[InlineData("MEGAFT^3", 0.02831684659, 0)]
[InlineData("DEG_C", 1, 273.15)]
[InlineData("M/S", 1, 0)]
[InlineData("MI/HR", 0.44704, 0)]
[InlineData("MI/KILOHR", 0.44704, 0)]
[InlineData("KG*M", 1, 0)]
[InlineData("LB*FT", 0.138254954376, 0)]
[InlineData("M*DEG_F", 0.5555555556, 0)]
[InlineData("DEG_C/HR", 0.00027777777777777777777777778, 0.0)]
[InlineData("M/DEG_C", 1.0, 0.0)]
[InlineData("BTU_IT/DEG_F*LB", 4186.8, 0.0)]
[InlineData("KILOMI^3/HR^2", 321.6189680123904, 0.0)]
```

Figure 26: Unit Test Data

The method shown in Figure 25 is then run with the data shown in Figure 26. The multiplier and the offset of each test has been obtained using https://www.wolframalpha.com/. The units have been chosen to cover all the different code paths, which can be seen in Table 1.

Table 1: Test data

Code path	**Test unit**
Base unit	M, S, and FT
Offset	DEG_C
Multiplication	KG*M, LB*FT, and M*DEG_F
Fraction	M/S, MI/HR, DEG_C/HR, and M/DEG_C
Multiplication and fraction	BTU_IT/DEG_F*LB
Prefix	KILOFT and PICOFT
Power	FT^2 and FT^3
Prefix and power	DECIFT^2 and MEGAFT^3
Fraction and prefix	MI/KILOHR
Fraction, prefix and power	KILOMI^3/HR^2

As can be seen from the code, the time to parse a unit was measured, but it quickly became apparent that it was so fast that any improvement would have miniscule influence on the system as a whole. The tick averages for 10000 repetitions of the parser can be seen in appendix 17.4. It took less than 100 ticks to parse even the most complex unit in the list above, where one tick is equivalent to 100 nanoseconds. This means that it took 10 microseconds to parse the unit. This is less than 1 % of the time it takes to convert and send one datapoint, as 10µs / 1ms = 0.01 = 1 %. Most of the time, processing a datapoint is even slower than 1ms, as can be seen in section 12.1. Furthermore, this operation is only done once, when a client first subscribes, which makes its performance even more insignificant.

13 Evaluation

The system manages to fulfill the basic goal of wanting to convert units in stream. It does this while making it easy, for both publishers of data and subscribers to the converted data, to use the system. The speed at which the system does this means that it could be integrated in bigger systems without slowing it down.

13.1 Functional requirements

The very first functional requirement states that one should be able to subscribe to any measurement stream. During this subscription, the system should check if the two units specified are compatible. This is done using dimension vectors. The system easily checks these, which makes sure that only compatible units are converted between, and thus the system does not send wrong data as a consequence of a user inputting an incompatible unit.

The system gets its data from a broker, and it was decided to use MQTT. MQTT has fulfilled the simple demands that the system has to it: It notifies the system whenever new data is posted, and it has a topic structure, which can be used in such a way that the system knows, what unit the posted data has, without it being preconfigured.

Finally, the system should be able to handle both prefixes and compound units. The system understands both standard SI prefixes and binary prefixes. It can also combine any two units, with or without prefixes, to any new unit. This means that the system can in theory work with any unit made up of any of the basic units already in the database. A beneficial consequence of this setup is that basic units can easily be added. They just need to be inserted into the database. The system does not even need to be rebuild or restarted.

13.1.1 Significant figures

A potential crucial point when dealing with units is the number of significant figures which is supported. The system uses the floating-point type in C# with the highest precision. This type is called decimal and has a precision of 28 digits. ”The binary representation of a Decimal value consists of a 1-bit sign, a 96-bit integer number, and a scaling factor used to divide the 96-bit integer and specify what portion of it is a decimal fraction” [9]. It can therefore accurately represent numbers like 0.1, which cannot accurately be represented by IEEE 754 floats [10]. This means that any number with more digits than 28 loses some of its precision. This is especially an issue with very large or very small numbers, because all units are converted to and from a base unit. All units are therefore very shortly stored as the base unit. Prefixes could be better utilised to support higher precision. How this could be done will be described in section 14.1 A single conversion object.

13.2 Performance measuring

Throughout the process of designing a way to measure the performance of the system we hit many snags. One of the first ones we hit was the delusion that the time to convert a single message would change over the load of the system, and after finding this to be false we were faced with the challenge of measuring end to end latency i.e., how long it would take for a reading to be sent through the entire system and finally arrive at the receiving client. In the process of developing this feature, we hit a snag; The client that produced data was initially hosted on a Raspberry Pi, whereas the server and test client were hosted on a desktop. The test relied on the fact that the two clocks were in perfect sync down to at most a millisecond, which was not something that was achievable as the windows desktop running the server and client had little control over the “Real Time Clock” (RTC

for short) in the system and the Raspberry Pi was completely without a RTC and as such highly inaccurate in its time keeping. To solve this issue, we simply moved the value producing MQTT client onto the desktop, to ensure that they were using the same RTC for their timestamps.
The next question to ponder for the performance measurement was at what frequency should messages be sent to the system. For this one we went with a range of frequencies specifically 1,2,5,10 Hz, as anything more than 10 Hz seemed excessive for measuring things in buildings, and 1 Hz as the less would result in the server exceeding the limit of 5000 client connections, before the messages delay would exceed the messaging frequency, and without exceeding the limits of the system, one cannot really say they know the real limits of the system.

We also made the choice to only do a single run of our test due to the large amount of time required to run the tests. Running the test for a single version of the server at a single frequency required a minimum of 2.5 hours, and the tests had to be scrapped if even a single connection was lost during the tests, as that would skew the numbers, though this apparent instability is an artifact of testing, and would have no real world impact, as its only when the connection is established, and a reconnect is easy to perform, it is just our testing that cannot tolerate these failed attempts. These tests therefore required human supervision, to avoid excessive time loss, and to stop the tests when completed. Another limiting factor was that the computer running the test could not be used for other things while running the tests, to avoid skewing the numbers by having different loads on the system at different times and during different tests.
All of this resulted in an excess of 30 hours spent monitoring a computer while the tests were running, as such making running multiple tests on the same settings an excessive amount of time to spend in our opinion.

13.3 Non-functional requirements

While it is important that the system functions correctly, the system also needs to fulfil its non-functional requirements to ever be considered for use. Through testing, it was shown that the system can handle over 1000 clients listening to data being posted at 5 Hz. The test was only done using a single server. The system can also easily be scaled up, both horizontally and vertically. Horizontal scaling is easier than with many other systems, as all the persistent data is only read and never written to. There is also no need to share data across clients, so server instances can work completely independent from each other. This test was performed in a worst-case scenario of all clients subscribing to only a single topic.

During the many hours of testing, the system has also proven its availability. It has never crashed, which speaks for its availability. However, the testing was mostly done in sessions of up to three hours. The system has run for many hours total, but it has been frequently restarted in between. If one were to properly assess the availability of the system, it would have to run for weeks on end.

13.4 ORM vs SQL

At the start of the project, it was believed that the database would only be accessed, when the system needed to look up a unit. This would be a single SQL-statement, which was easy to write. While this is true, it makes any future modification involving the database more of a hassle. Being able to do set operations directly on a set of objects will always be faster and easier than writing SQL statements. Another argument for using SQL statements is that the developers are more in control in case of poor performance for more advanced queries. This never ended up applying, as the system only does very simple select queries. All in all, the decision to use pure SQL was poor in hindsight and based on a wrong belief that a single SQL query is easy to just implement. An ORM, like .NET's entity framework [11], would probably have been just as easy, if not easier.

14 Future work

14.1 A single conversion object

A central improvement would be to create a single conversion object. This would combine the from unit and the to unit objects into a single object. This would save a multiplication and an addition. More importantly, it would better utilize prefixes, as a way to improve the precision of numbers. As the system is currently, all data is converted to the base unit without prefixes, before it is then converted to the desired unit and the prefix is multiplied with the number. This means that any mathematical advantages gained from prefixes is lost, and it is purely a tool to make the number easier to understand for the user.

For example, if one where to convert angstrom to picometres, the system, as it is now, would first convert the value to metres, before it would then convert it to picometres. This means that it only supports 28 (precision of C# decimal object) - 10 (conversion ratio between angstrom and metres) = 18 significant figures. However, if the system used a single conversion object, it would not have this intermediate step and could keep all its 28 significant figures.

14.2 Caching units for faster subscription

One of the features that could be implemented in the system to speed up the time it takes to subscribe to a conversion, could be the caching of conversions. Let say that you subscribe to the conversion of one sensor from Fahrenheit to Celsius, and the later want to subscribe to a different sensor with the same conversion of ºF to ºC, a caching of the conversion would allow the system to skip the step of querying the database for the two units and constructing a conversion, thereby making the system a tiny bit faster. We chose not to implement this, as per 8 Requirements the subscription only happens once per connection per client and as such any performance improvement here would be negligible from an overall perspective.

14.3 Implementing support for other stream solutions

Implementing interfaces in the system to support other stream solutions, such as Kafka, would improve the utility of the system greatly as many building automation systems use Kafka to handle the data streams. Implementing the interfaces for supporting other stream solutions would also make it significantly easier to implement other stream solutions as opposed to directly coding support for Kafka specifically. The current stream client interface reflects the intention to use MQTT and might need further generalization to be suitable for other stream solutions.

15 Conclusion

Units are ever present, and the need to convert between them will always exist. A simple and quick way to convert units would be beneficial to a lot of people. Being able to convert units in stream is essential for the system to be a viable choice in many cases. IoT is becoming more and more widespread and with it comes a constant stream of data. This data needs to be processed and acted upon. To be able to properly act upon it, it needs to be in the right units. As units are tied closely to culture and are inherently diverse, the system needed to be flexible and easily updatable.

This has been achieved by the system having the capability of compounding units together to form any unit. This means that all units do not need to be added to the database. If a unit consists of two or more other units, which the system knows about, the system can convert from and to it, without having it in the database. At the same time, if the need arises for a new unit to be added to the database, the process is a very simple SQL insert statement. The system does not even need to be restarted, as it queries the database whenever a new subscription occurs. This contrasts with QUDT, where every unit has to be defined. QUDT is much less flexible.

With potentially being used in an IoT context comes high demands for through put and availability. Most IoT systems are designed to run 24/7, which, of course, reflects the high importance put on availability. The system has been tested to ensure that it lives up to these demands. It has the capability of handling 1000 clients listening to data being posted at 5 Hz or in other terms it has the ability to handle a message throughput of 5000 messages pr second. According to Aslak, building OU44 at SDU has upwards of 80000 data providers with an average posting frequency of <1/min. This comes out at 80000/60=1333 messages per second, which is much lower than our systems throughput. Furthermore, during this testing the system has not crashed once.

16 References

[1] G. Novak, "Conversion of units of measurement," *IEEE Transactions on Software Engineering,* vol. 21, no. 8, pp. 651-661, 1995.

[2] QUDT, "QUDT," QUDT, [Online]. Available: https://www.QUDT.org/. [Accessed 26 05 2021].

[3] Wikipedia, "Internet of things," 26 02 2021. [Online]. Available: https://en.wikipedia.org/wiki/Internet_of_things.

[4] Nist, "Nist physics," [Online]. Available: https://physics.nist.gov/cuu/Units/binary.html. [Accessed 26 03 2021].

[5] Bureau International des Poids et Mesures (BIPM), "si_brochure_8_en.pdf," 26 02 2021. [Online]. Available: https://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf.

[6] IEC/ISO, »IEC 80000-13,« IEC/ISO, 2008.

[7] Microsoft, "Performance," Microsoft, 20 12 2012. [Online]. Available: https://github.com/SignalR/SignalR/wiki/Performance. [Accessed 25 05 2021].

[8] Apache, "https://kafka.apache.org/," 01 05 2021. [Online].

[9] Microsoft, "Decimal Struct," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.decimal?view=net-5.0. [Accessed 27 05 2021].

[10] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008),* pp. 1-84, 2019.

[11] Microsoft, "Entity Framework Documentation," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/ef/. [Accessed 25 05 2021].

[12] C. J. Wells, "technologyuk.net," 30 06 2016. [Online]. Available: https://www.technologyuk.net/science/measurement-and-units/physical-quantities-and-si-units.shtml. [Accessed 25 03 2021].

[13] P. Fletcher, "microsoft.com," Microsoft, 06 10 2014. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr. [Accessed 05 04 2021].

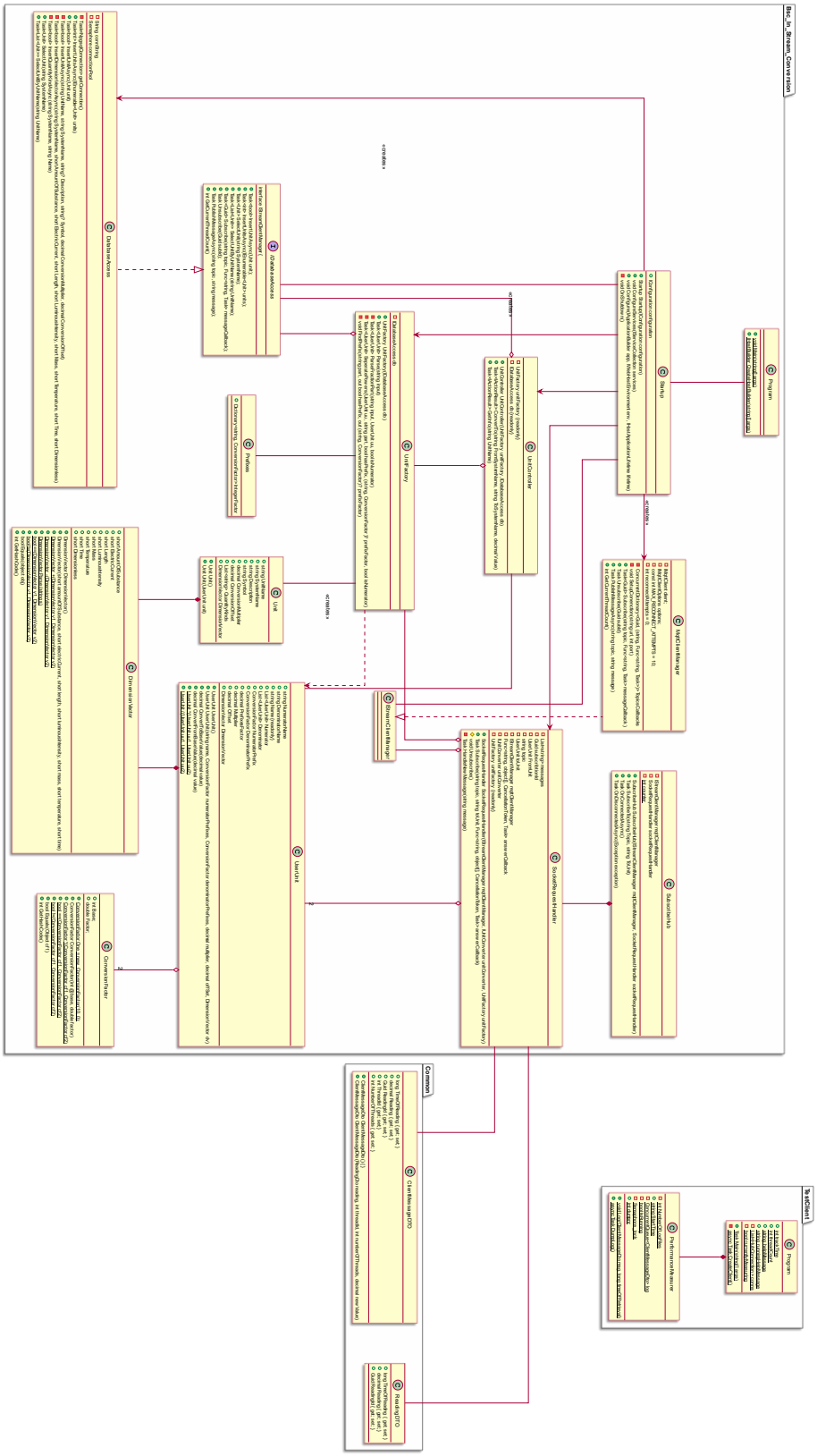
[14] Microsoft, "microsoft.com," Microsoft, [Online]. Available: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core. [Accessed 05 04 2020].

[15] Microsoft, "Task-based asynchronous programming," Microsoft, 30 03 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming. [Accessed 24 05 2021].

17 Appendix

17.1 Class diagrams

17.1.1 Complete diagram



17.2 Use case specification

Use case: Look up units
ID: 01
Brief Description: The potential subscriber looks up the internal name of a unit
Primary Actors: Subscriber
Secondary Actors: N/A
Preconditions: Desired unit exists in the system
Main flow: 1. The subscriber enters the name of the unit into the system. 2. The system responds with all possible matches.
Postconditions: The user knows of their desired unit
Alternate flows: N/A

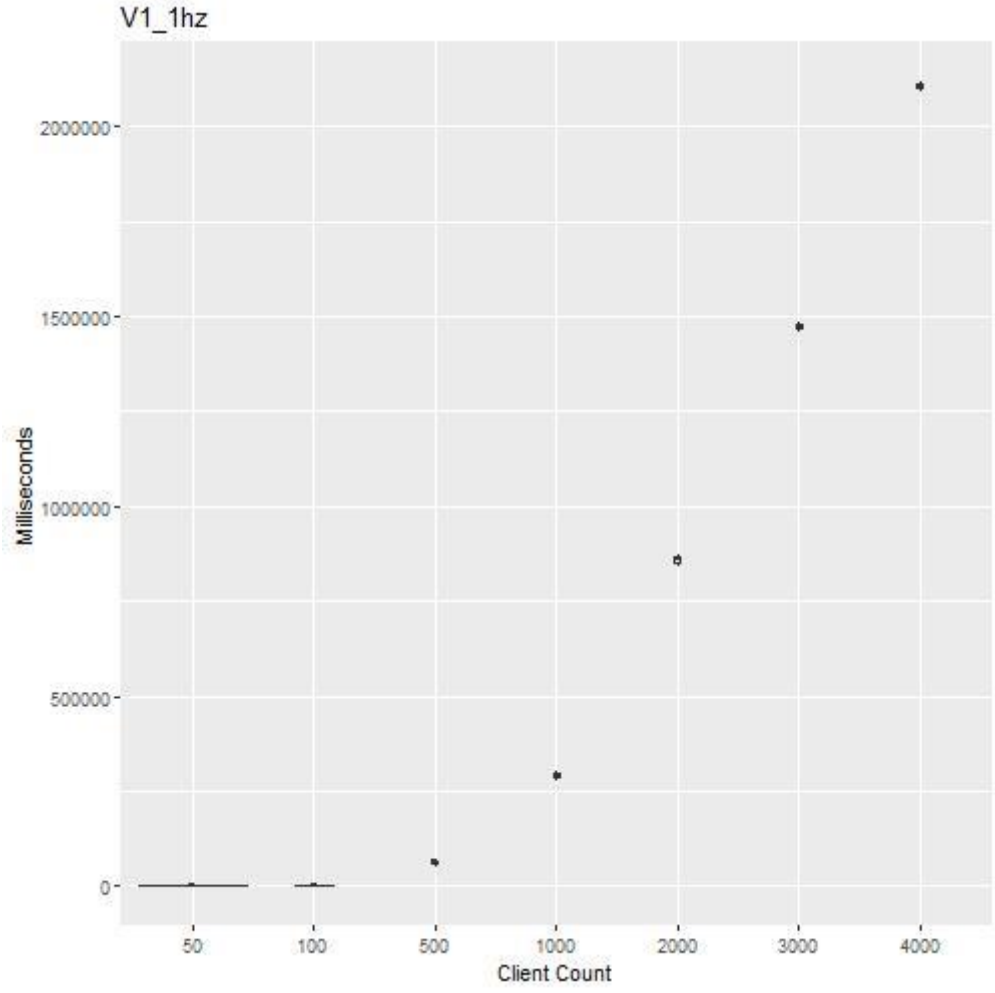
Use case: Subscribe to system
ID: 02
Brief Description: The subscriber subscribes to the system specifying the desired return unit
Primary Actors: Subscriber
Secondary Actors: N/A
Preconditions: 1. Know of system unit name. 2. Broker online.
Main flow: 1. Send subscribe request containing desired unit and broker details. 2. System validates the unit. 2.1. **If** validation fails. 2.1.1. Return error. 2.1.2. End flow. 3. System connects to broker.
Postconditions: 1. Subscriber is connected and ready to receive data. 2. System is listening for update from the broker
Alternate flows: N/A

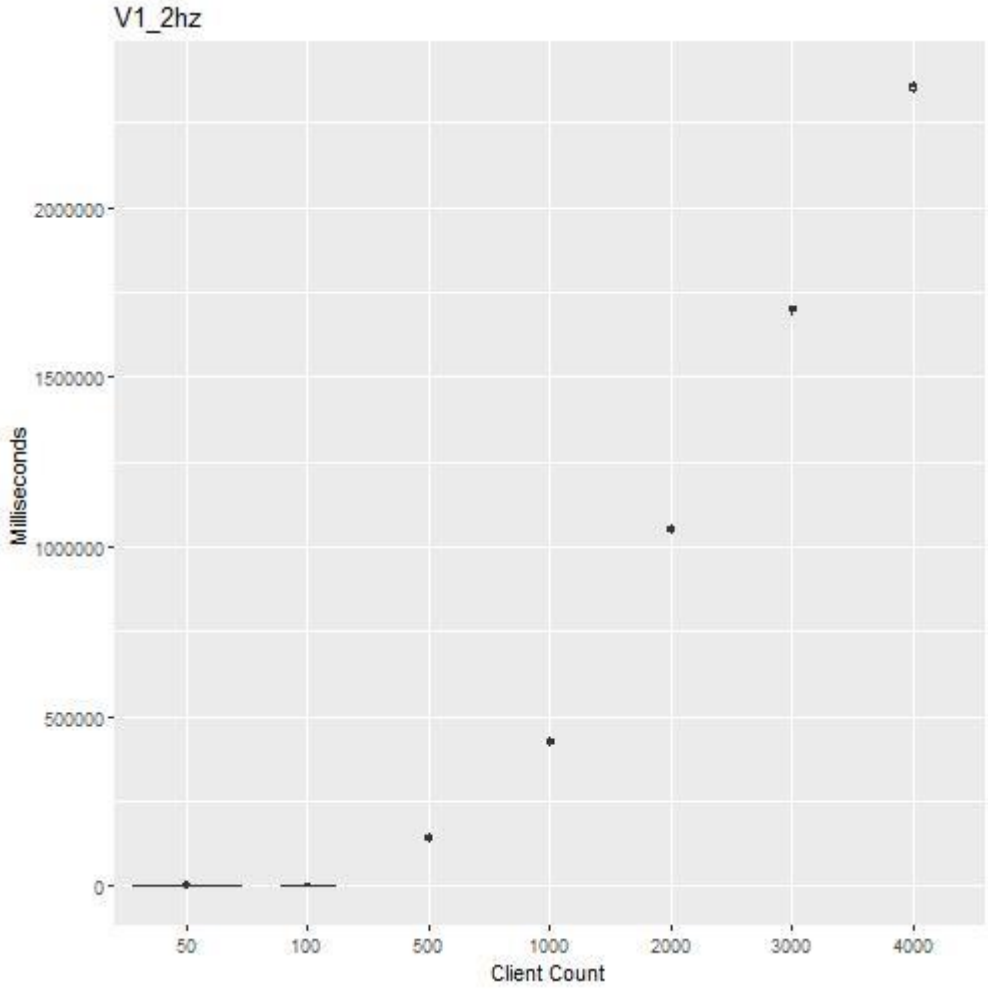
Use case: Post data to broker
ID: 03
Brief Description: A data provider published data to the broker
Primary Actors: Data Provider
Secondary Actors: N/A
Preconditions: Broker online
Main flow: 1. Provider connects to broker. 2. Provider publishes data to broker.
Postconditions: Data send to any subscribed clients[2]
Alternate flows:

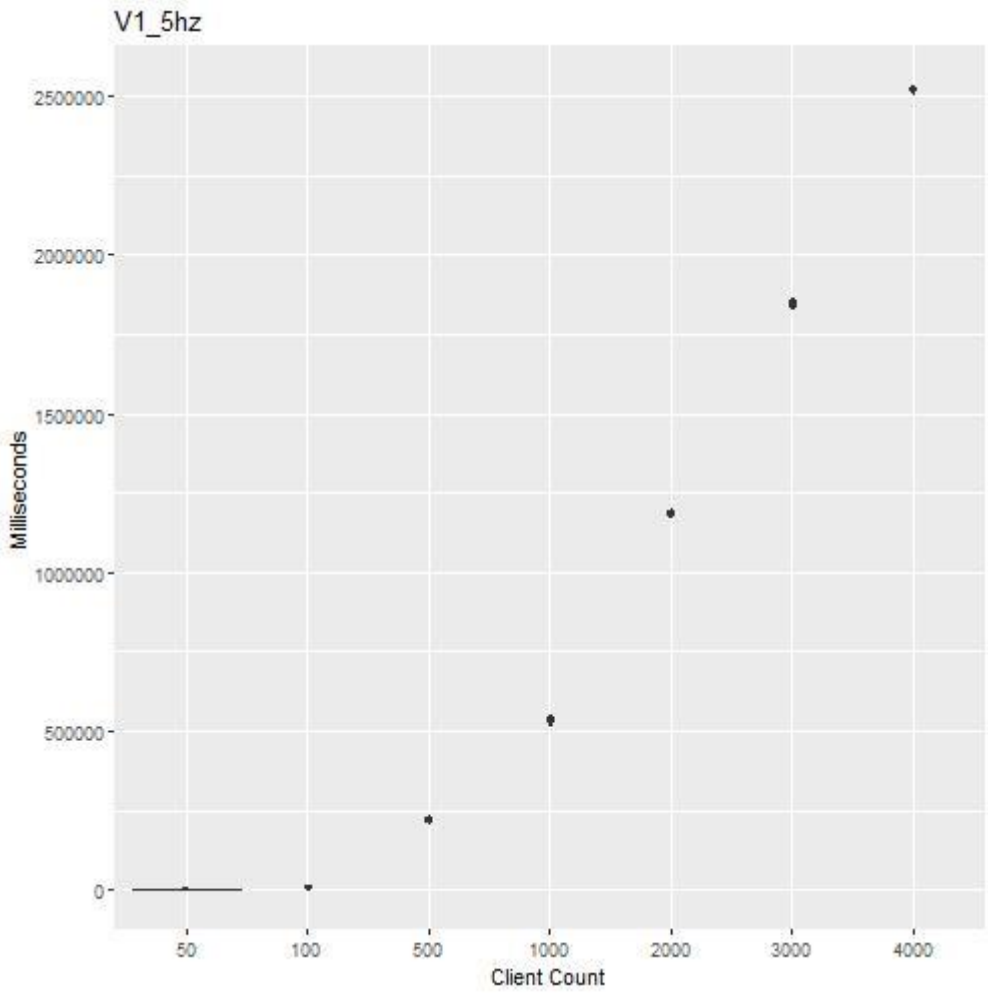
Use case: Receive data
ID: 04
Brief Description: The server receives data from a broker that at least one subscriber has subscribed to
Primary Actors: Subscriber
Secondary Actors: N/A
Preconditions: Subscriber has subscribed. Data has been published
Main flow: 1. The server receives data from a broker. 2. The server looks at all subscriptions for the data. 3. **For each** subscriber subscribed to the data. 3.1. System converts data. 3.2. System sends data to subscriber.
Postconditions: N/A
Alternate flows: N/A

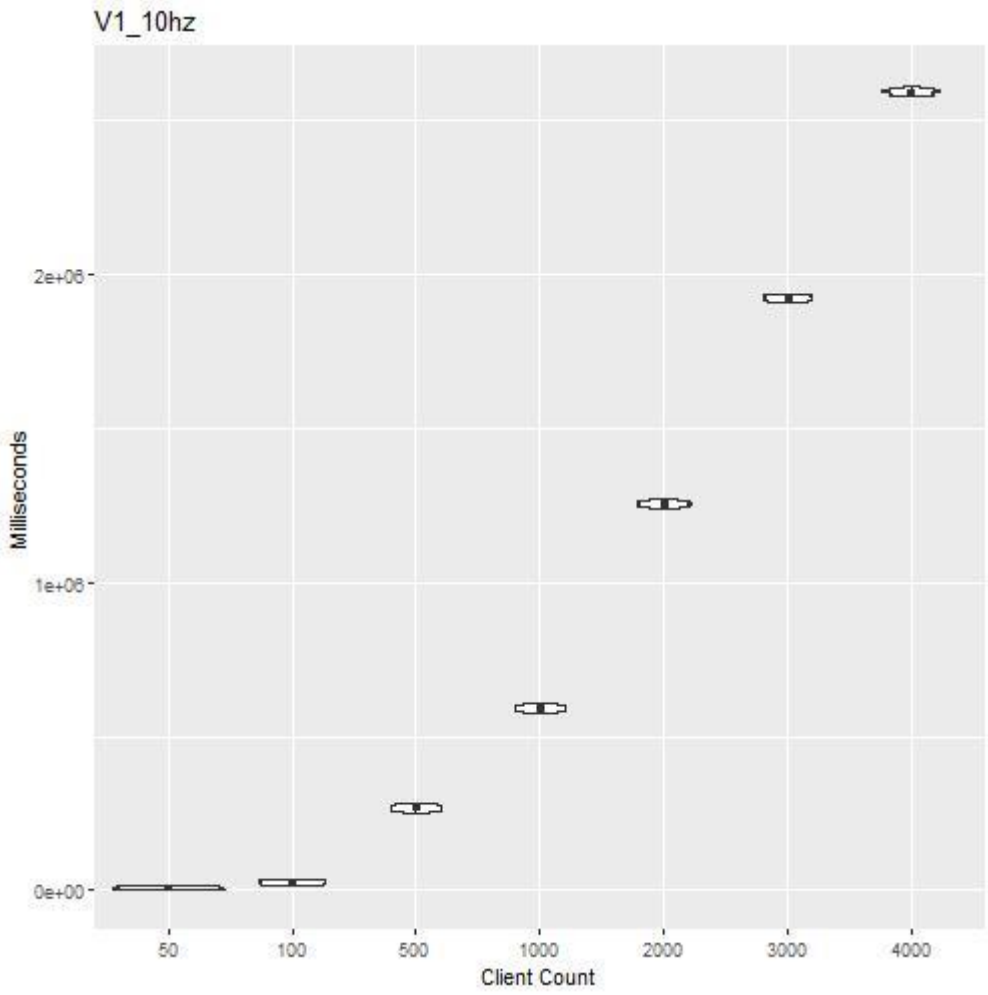
[2] This refers to clients subscribed to the broker and not the actor called subscriber.

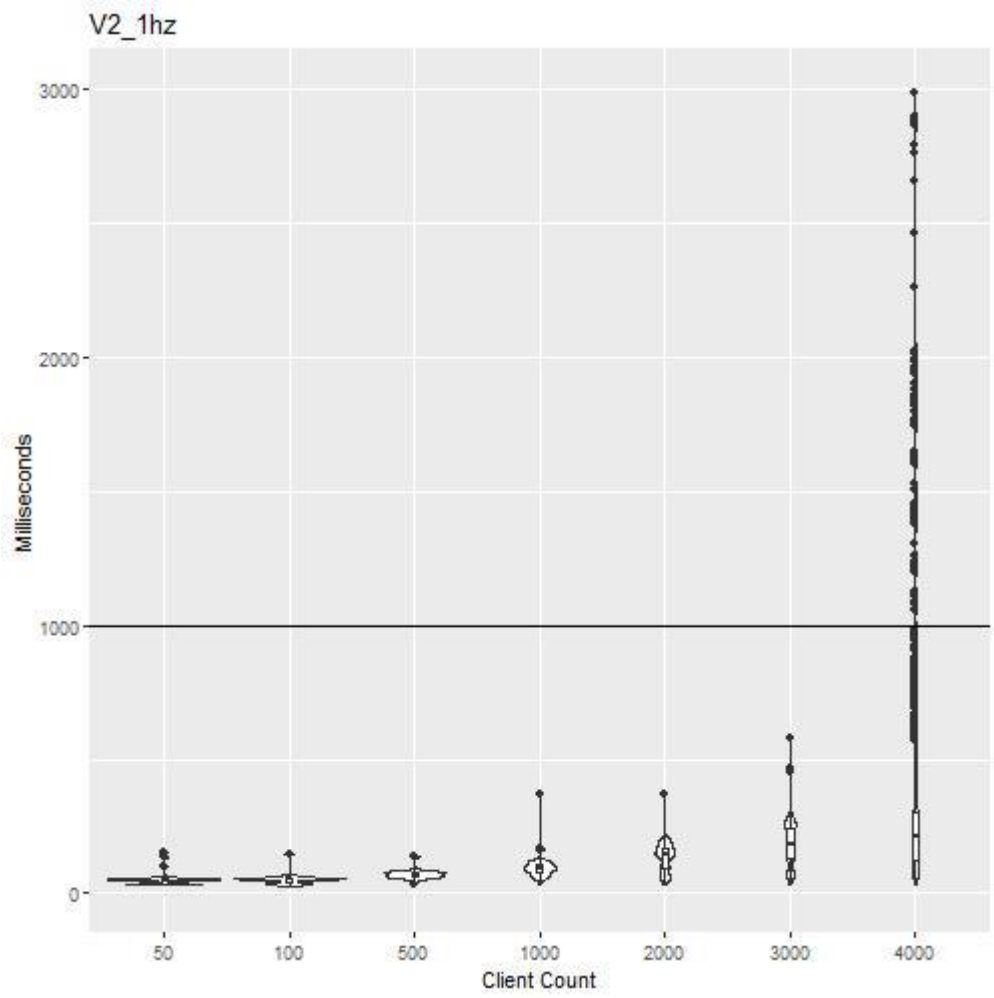
17.3 Performance Measurements

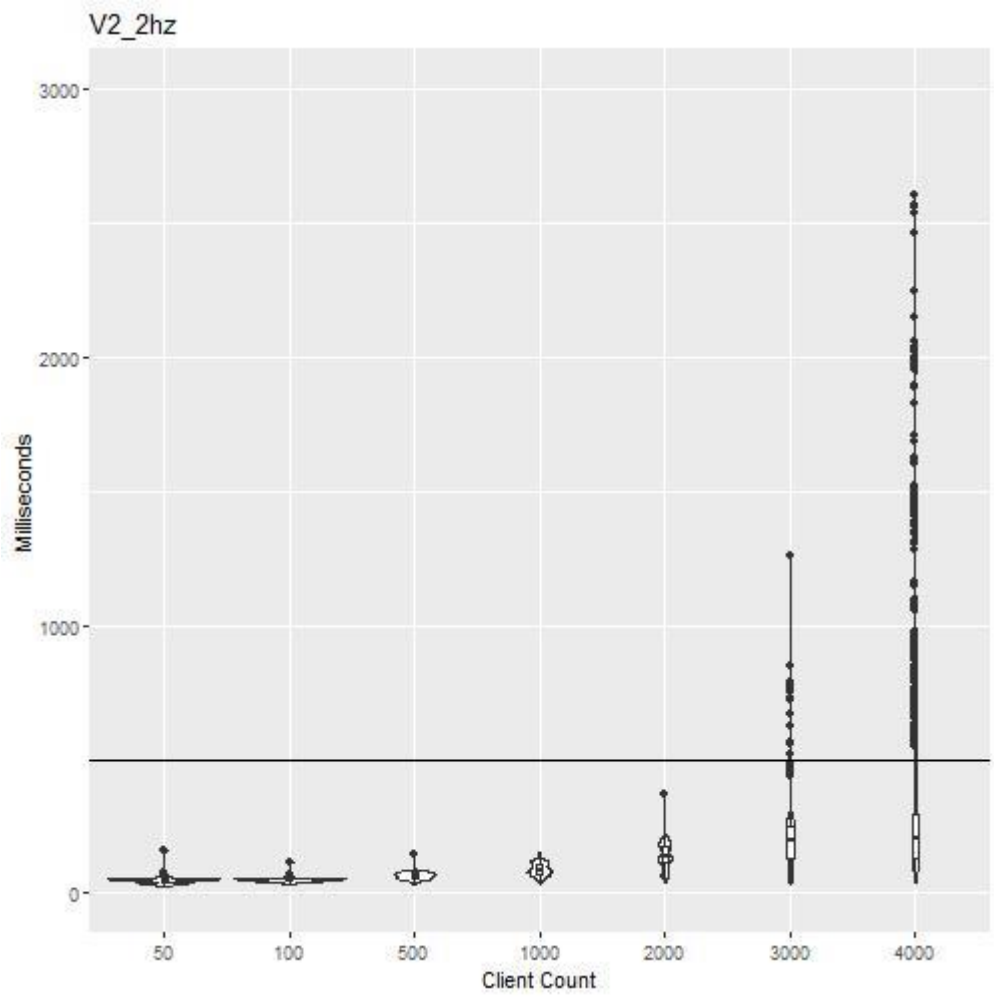


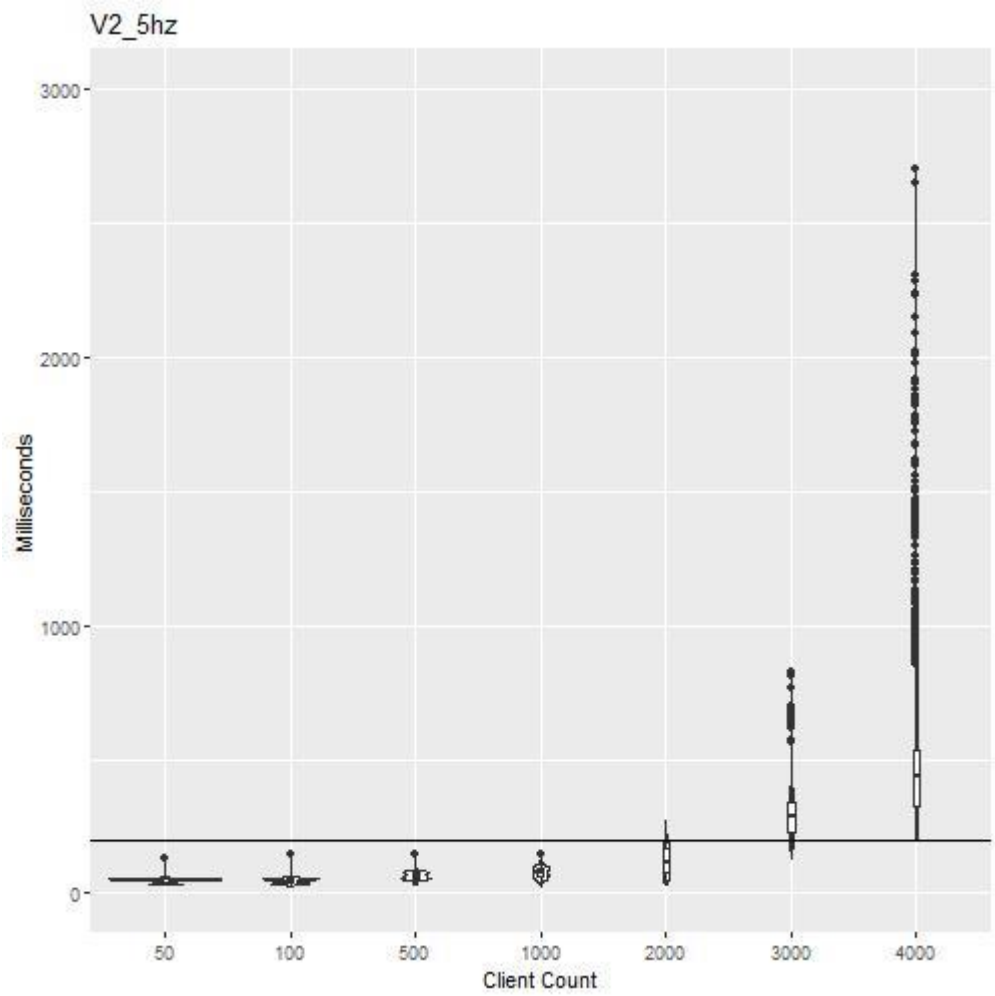

V1_2hz
2000000
1500000
1000000
500000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count

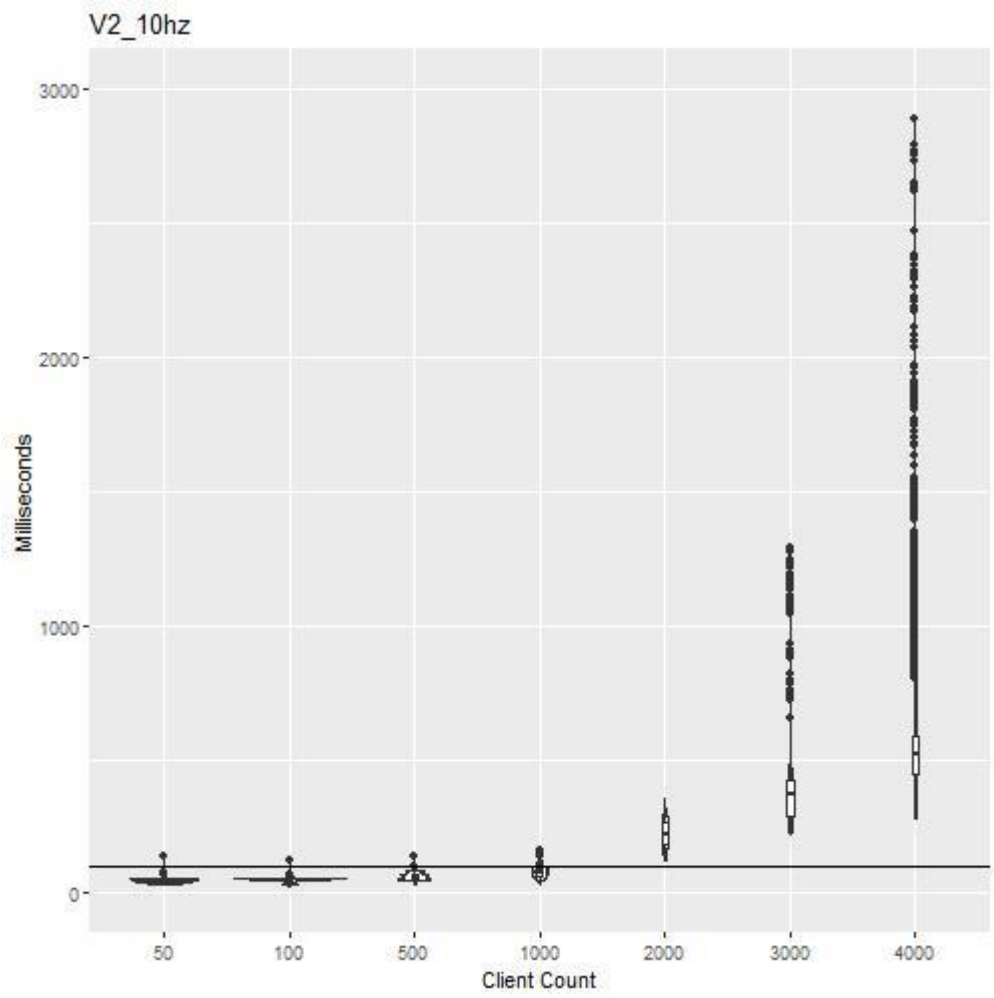

V1_5hz
2500000
2000000
1500000
1000000
500000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count

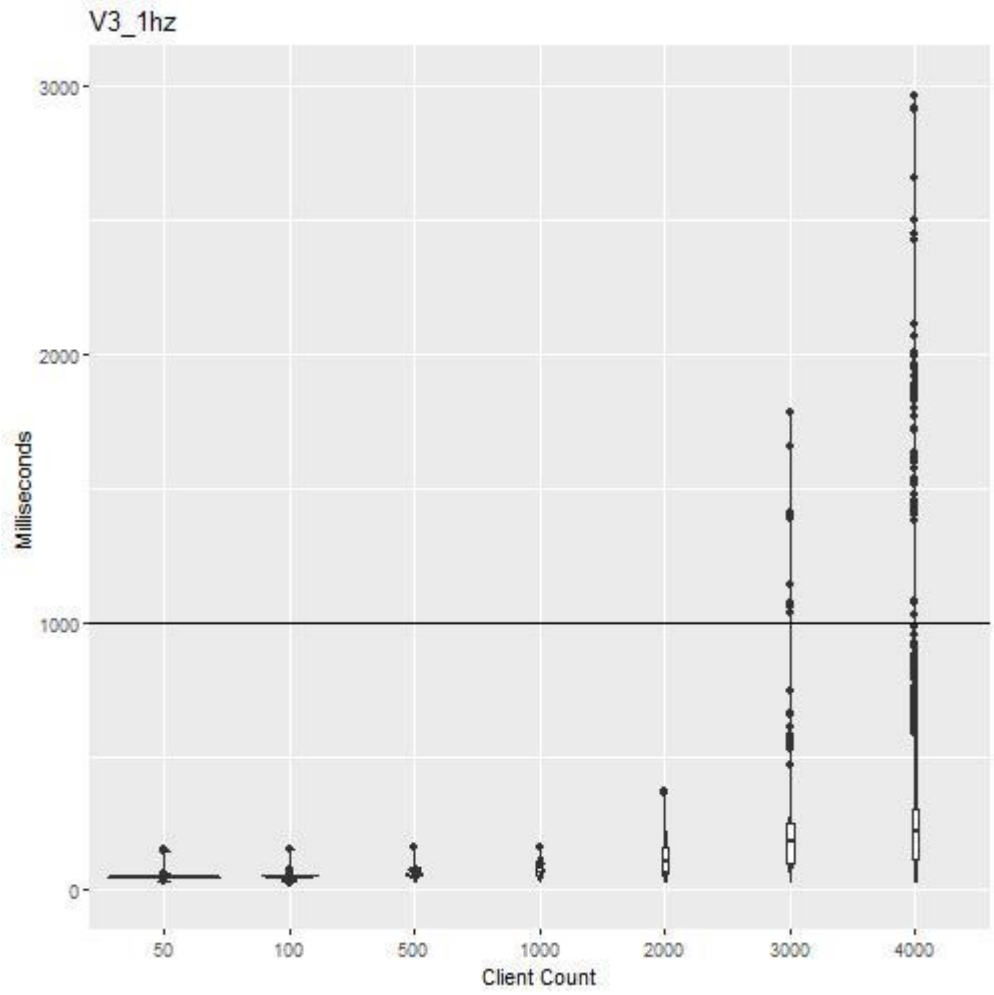

V1_10hz
Milliseconds
2e+06
1e+06
0e+00
50
100
500
1000
2000
3000
4000
Client Count

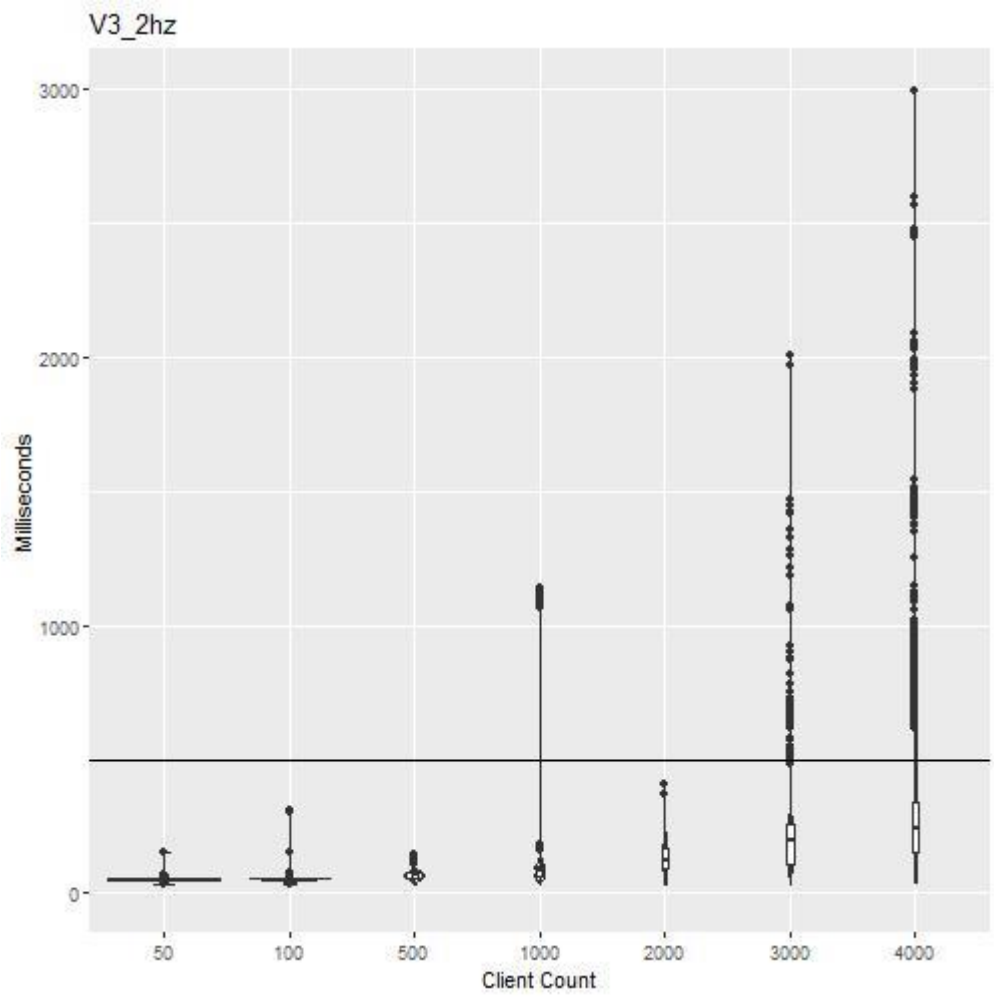

V2_1hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count

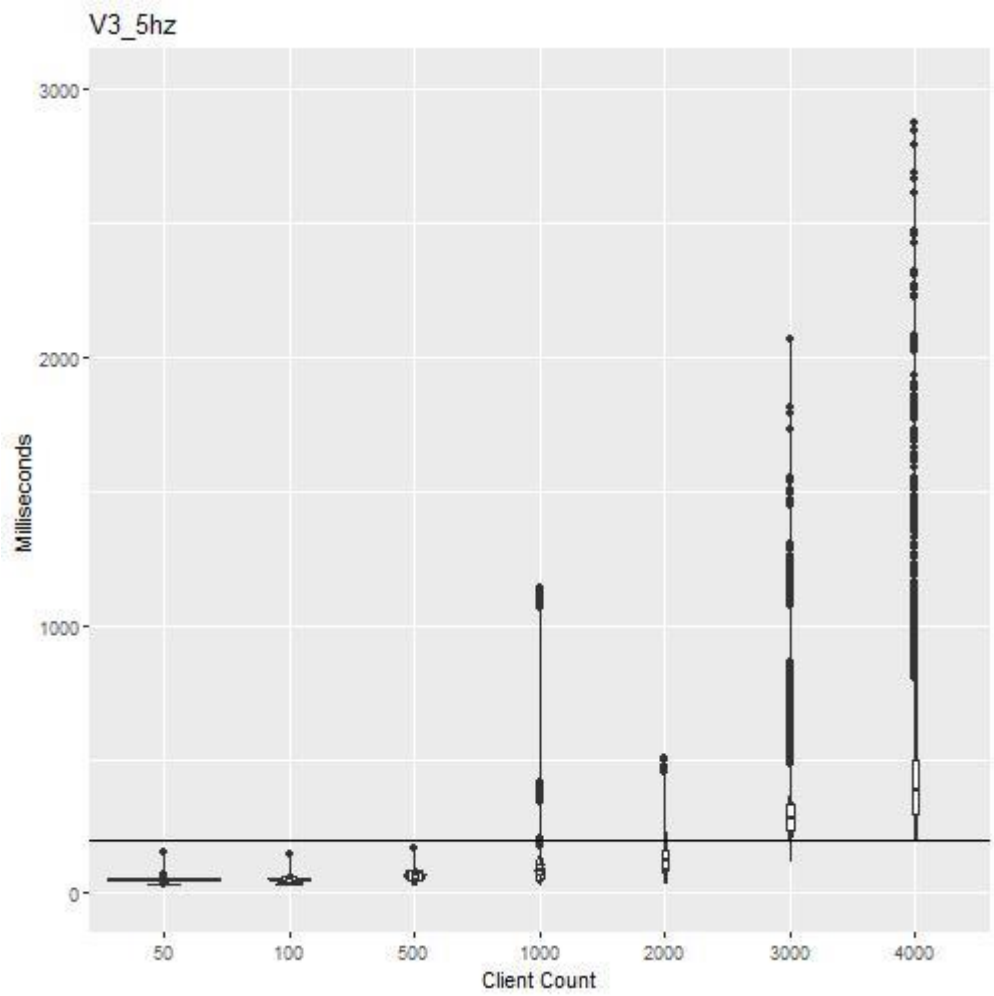

V2_2hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count

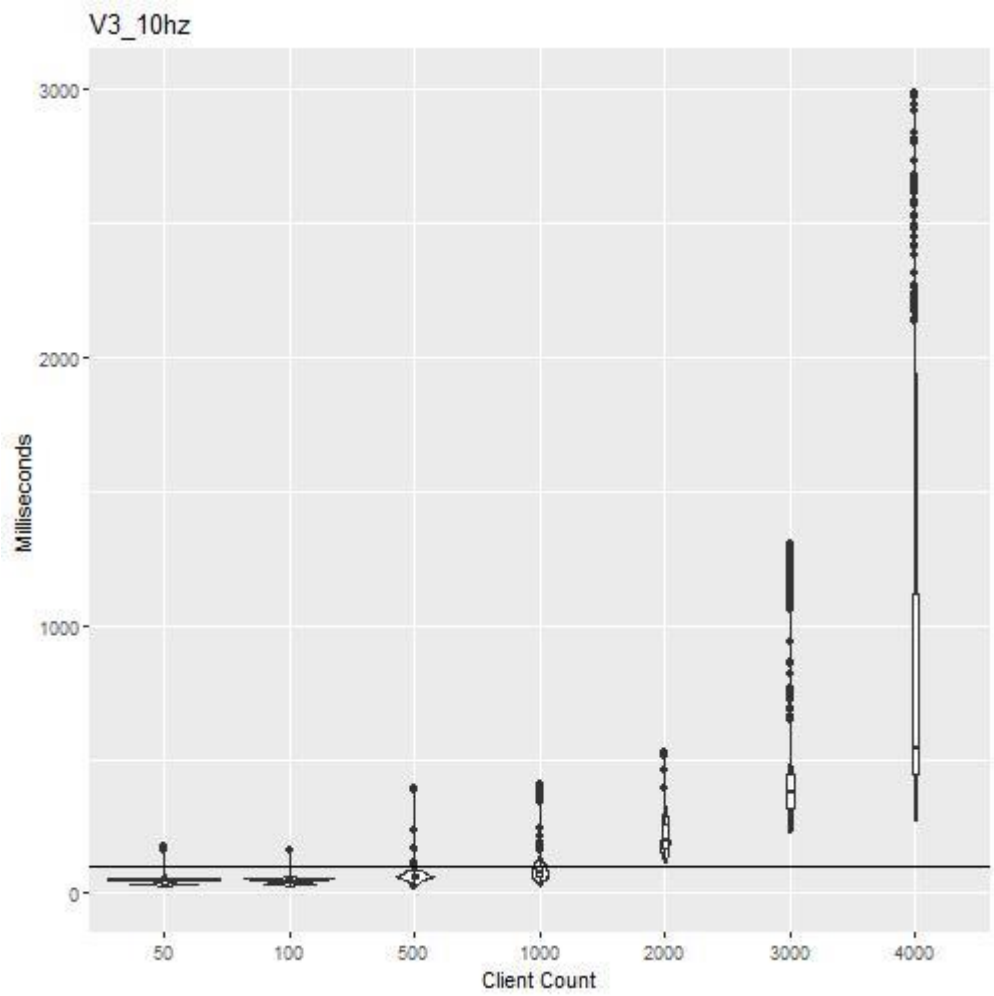

V2_5hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count


V2_10hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count


V3_1hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count


V3_2hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count


V3_5hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count


V3_10hz
3000
2000
1000
0
Milliseconds
50
100
500
1000
2000
3000
4000
Client Count

17.4 Parser performance

UNIT	TICKAVERAGE
S	14.60
KILOFT	15.46
PICOFT	16.59
M	17.48
FT	18.00
DEG_C	22.56
FT^2	30.20
DECIFT^2	33.99
MI/HR	40.00
LB*FT	45.53
FT^3	47.36
MEGAFT^3	47.40
DEG_C/HR	47.55
KG*M	48.09
M*DEG_F	48.13
KILOMI^3/HR^2	83.68