Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Simen Norderud Jensen

Building an extensible prototype for a cloud based digital twin platform

Master's thesis in Engineering and ICT
Supervisor: Terje Rølvåg
June 2019

NTNU
Norwegian University of
Science and Technology

Simen Norderud Jensen

Building an extensible prototype for a cloud based digital twin platform

Master's thesis in Engineering and ICT
Supervisor: Terje Rølvåg
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Abstract

Digital twin solutions for structural integrity monitoring and predictive maintenance is a young and promising field, and there has been an increasing amount of publications on it. There is still much to be desired when it comes to case-studies though, and affordable offerings for Digital Twin development are currently lacking.

This thesis aims to lay the groundwork for a platform that could serve the ability to both develop and use Digital Twins as a cloud computing resource. This is done by developing and documenting an extensible prototype from a carefully chosen technology stack. The prototype developed in this thesis, named Tvilling Digital, is meant to be a stable and extendable platform that can be built upon in later theses. It uses a self-developed system, named the blueprint system, to enable easy additions of custom filters, analyzers, solvers, and any other tool that takes inputs and returns results. Support for Functional Mock-up Units and some simple filtering and spectral analysis tools has been added through this blueprint system during the project.

In addition to the choice of tools and the development of the prototype, this thesis will also discuss how the prototype can be further developed to achieve a full-fledged solution.

Sammendrag

Løsninger som benytter Digitale tvillinger for strukturell integritetsovervåking og prediktivt vedlikehold er et ungt og lovende emne, og det har vært en økende mengde publikasjoner innenfor det. Det er allikevel kun noen få av disse som er saksstudier, og det er fortsatt en mangel på rimelige løsninger for utvikling av digitale tvillinger på markedet.

Denne oppgaven prøver å legge grunnlaget for en plattform som kan tilby muligheten til å både utvikle og bruke digitale tvillinger som en skyressurs. Dette gjøres ved å utvikle og dokumentere en utvidbar prototype fra nøye valgt teknologi. Prototypen utviklet i denne avhandlingen, kalt Tvilling Digital, er ment å være en stabil og utvidbar plattform som kan bygges på i senere oppgaver. Den bruker et selvutviklet system, kalt blueprint-systemet, for å muliggjøre enkle tillegg av tilpassede filtre, analyseverktøy, løsere og andre verktøy som tar imot data og returnerer resultater. Støtte for Functional Mock-up Unit og noen enkle filtrerings- og spektralanalyseværktøy har blitt lagt til gjennom dette systemet i løpet av prosjektet.

I tillegg til valg av verktøy og utvikling av prototypen, vil denne oppgaven også diskutere hvordan prototypen kan videreutvikles for å oppnå en fullverdig løsning.

Contents

List of Listings

List of Figures

Glossary

FEDEM A computer program for multibody simulation of mechanical systems. 1, 7, 9, 10, 13, 28, 31, 33, 35, 36

Tvilling Digital The solution created in this thesis. 2, 7, 9, 10, 16–18, 22, 23, 27, 31, 33–36, 38

WebSocket A computer communications protocol supported by most modern browsers. 9, 10, 16

Acronyms

API Application Programming Interface. 2, 7, 9, 11, 16, 17, 23, 27, 30, 34

DAQ Data AcQuisition system. 13

DT Digital Twin. 1, 2, 4–9, 13, 17, 18, 36, 38

FE Finite Element. 1, 18, 35, 36

FEM Finite Element Method. 8

FFT Fast Fourier Transform. 1

FMU Functional Mock-up Interface. 5–8, 10, 13, 22, 23, 28, 31–36, 38

LSSA Least-Squares Spectral Analysis. 1, 22

MVP Minimum Viable Product. 10

PoC Proof of Concept. 1, 7–9

UDP User Datagram Protocol. 13

1 Introduction

1.1 Background and motivation

Digital Twin (DT) solutions for structural integrity monitoring and predictive maintenance is a young and promising field. There has been an increasing amount of publications around both DTs and DT solutions for structural integrity monitoring and predictive maintenance. Only a few of these were case-studies though, and only one of the case-studies found tackled real-time Finite Element (FE) simulations. Affordable DT solutions for structural integrity monitoring are also lacking at the moment. Many large companies, like SAP, GE, SIEMENS, and ANSYS, have invested in DT, but their current solutions are expensive, proprietary, and in some cases reliant on their proprietary ecosystem.

A Proof of Concept (PoC) DT solution for monitoring a torsion bar suspension rig using FEDEM on a cloud computing resource was developed as a part of the specialization project last year. The PoC consisted of a physical torsion bar suspension rig, a DT of the torsion bar suspension rig, and a web frontend for visualizing the data from the DT in real-time. The DT consisted of a FE model being simulated in FEDEM and receiving data from the test rig. The torsion bar suspension rig, which is also used in this project, is described in section 3.2.2.

The solution created last year performed well but was tailormade for the torsion bar suspension rig. It would, in other words, be necessary to create a new system for every unique DT using this method. While much of the system could be reused, it would still require a significant amount of unneccesary work each time. There were also several desirable features that would have made the prototype system considerably more useful.

One of the desirable features was integrated filtering of the incoming data. Noise in the incoming sensor data can create problems with the FE simulation. Highpass and lowpass filtering, for example via a Butterworth filter, would be extremely helpful for dealing with common noise factors like electromagnetic interference, which is often periodic. Another desirable feature was integrated frequency analysis. Frequency analysis, in the form of for example a simple version of Fast Fourier Transform (FFT) or the more advanced Least-Squares Spectral Analysis (LSSA), would open the way for a more advanced understanding of the twins' behaviour. Among the other desirable features were the ability to trigger actions on specific events and generate reports from old data.

I wanted to create the groundwork for a platform that could host multiple different DT for multiple different users with different needs. Therefore I did not want to just create tailormade versions of the aforementioned desirable features and hard-code them into the platform. Instead I decided to develop a generic way to integrate features like these into the platform easily, without having to change the platform itself. Users can then decide between using existing filters, solvers, etc. with their own parameters or create their own specialized filtering, spectral analysis, etc. algorithms and seamlessly integrate them in their DT system on the platform. To achieve this I defined the blueprint system described in section 3.3.5.

A proper user interface with more flexibility in the visualization of the data was also desirable. In this thesis, I decided to create the Application Programming Interface (API) necessary for a frontend instead of adding the creation of a frontend to the project scope. A frontend, in the form of a web application, was created for the platform using this API in Sande and Børhaug 2019. A proper interface with more flexibility in the visualization is among the features in this web application.

1.2 Problem description

The goal of this master's thesis is to lay the groundwork for a platform that could serve the ability to both develop and use Digital Twins as a cloud computing resource.

This will be accomplished by

1. Investigating relevant technologies and defining a technology stack for the platform.
2. Developing an extensible prototype, Tvilling Digital, using the chosen technology stack.
3. Documenting Tvilling Digital.
4. Discussing how Tvilling Digital can be expanded on to achieve the aforementioned platform.

1.3 Outline

This section will give a brief outline of the overall structure of the thesis.

Background contains relevant background theory used in the thesis.

Method contains description of and discussion around the methods used. The section starts by discussing the choices for the technology stack and describing the chosen technologies. It continues by describing the prototyping tools used. The overall development and documentation of Tvilling Digital is then described. The section ends by describing how the results in the results section are generated.

Results contains the results generated.

Discussion contains discussion about the results of the development and future work. The section starts by discussing the results of the choices done regarding the technology stack. It continues by discussing how the prototyping tools and the cooperation with other master students impacted the development. The results in the results section are then discussed. The section ends with a discussion on possible future work.

Summary gives a short summary of the thesis.

2 Background

This thesis mainly concerns itself with creating a solution for running DTs. The inner workings of specific types of twins, for example the FE simulations done using the inverse method in the DT for the Offshore Crane, is not addressed. Though it was necessary to create some filtering, wave generation, and spectral analysis tools to test with and use as examples for the blueprints described in Section 3.3.5, the theory behind their implementation is not relevant to the goal of this thesis and is therefore not included.

2.1 Digital Twins

Digital twins were classified as one of the top ten most promising technological trends in the next decade by Gartner in 2017 and 2018 and is drawing increasing attention in the academia (Tao et al. 2019). It is described as an important part of the Industry 4.0 technologies by several publications, such as (Roser 2015; Haag and Anderl 2018; Uhlemann, Lehmann, and Steinhilper 2017; Padovano et al. 2018; Schroeder et al. 2016; Grieves 2014; Tao et al. 2019; Ayani, Ganebäck, and Ng 2018; Negri et al. 2019; Schleich et al. 2017; Borodulin et al. 2017; El Saddik 2018) There are currently many different understandings of what a digital twin actually is though, as described by for example Schleich et al. 2017; Kritzinger et al. 2018; Tao et al. 2019.

According to Schleich et al. 2017 the first definition was probably made by NASA in their integrated technology roadmap, Shafto et al. 2010, TA11-7. It was described as

> an integrated multi-physics, multi-scale, probabilistic simulation of a vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its flying twin.

This definition was only slightly adapted to

> an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin

in Glaessgen and Stargel 2012. Not everyone agrees with this definition though.

Kritzinger et al. 2018 classified the use of the term "digital twin" in current literature into three types based on the level of data integration between the physical asset and digital representation in the described digital twin. Digital Model (DM), Digital Shadow (DS), and Digital Twin (DT). When there is no automatic real-time data communication between the physical asset and the digital representation, as in Figure 1, then the described digital twin is instead classified as a "digital model". When there is automatic real-time communication from the physical representation to the digital twin but not from the digital representation to the physical asset, as in Figure 2, then the described digital twin is classified as a "digital shadow". Only when there is an automatic real-time communication both from the physical asset to the digital representation and from the digital representation to the physical asset, as in Figure 3, is the described digital twin classified as a proper digital twin.

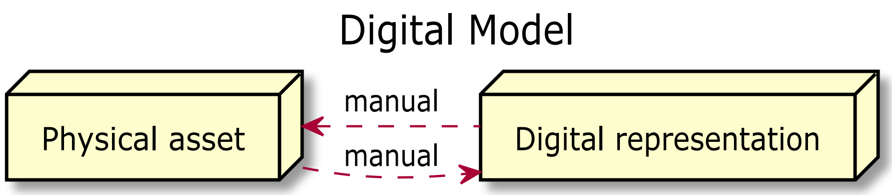


Figure 1: Data flow in a Digital Model according to Kritzinger et al. 2018

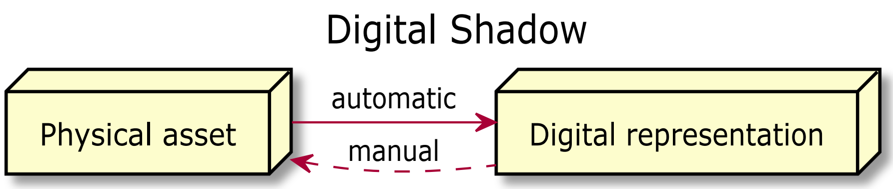


Figure 2: Data flow in a Digital Shadow according to Kritzinger et al. 2018

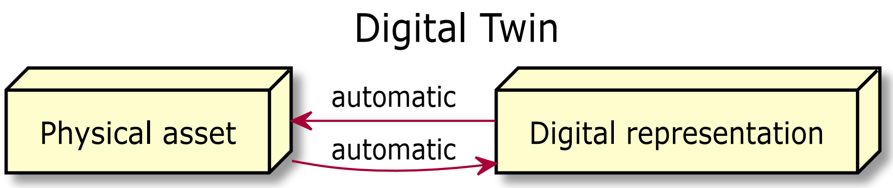


Figure 3: Data flow in a Digital Twin according to Kritzinger et al. 2018

This thesis will use the Glaessgen and Stargel 2012 definition of a DT, but extending the solution to supporting the type of DT described in Kritzinger et al. 2018 is discussed in future work.

2.2 FMI and FMUs

The Functional Mock-up Interface (FMI) is a standarized interface for dynamic models. FMI is supported by over 100 tools according to fmi-standard 2019b. Models following the FMI standard are called Functional Mock-up Units (FMU). By creating an FMU from a model it is possible to import them into FMI compatible programs regardless of the original format of the models. This is the "model exchange" part of the FMI standard. In addition to "model exchange" some FMUs also support "co-simulation". Functional Mock-up Interfaces (FMUs) supporting "co-simulation" also includes a solver for the model. These FMUs can be simulated independently in any co-simulation compatible program as long as the platform is supported by the solver binaries included in the FMU. The FMU blueprint is used to simulate using these co-simulation compatible FMUs.

2.3 Cloud computing

Cloud computing is a model that enables on-demand availability of computing resources (Mell, Grance, et al. 2011, p. 2).

2.3.1 Service models

Popular service models for cloud computing are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). SaaS gives the users access to the cloud provider's applications running on the providers cloud infrastructure. In PaaS the users are instead providing the applications themselves, while in IaaS they are also managing fundamental computing resources themselves. A further developed version of Tvilling Digital could potentially provide DT development and use to users using the SaaS model, possibly in a similar way to the Digital Twin as a Service (DTaaS) model described in Borodulin et al. 2017.

2.3.2 Scalability

Scalability is an important part of cloud computing. In cloud computing there are two ways for a software to scale:(Furht and Escalante 2010, p. 358)

1. Vertical scaling: by increasing the resources of the node(s) the software runs on.
2. Horizontal scaling: by increasing the number of nodes the software runs on.

Scalability is important if the software is to be used by a large number of people, for example if it is used to create a SaaS. It can also be important for reducing the resources used. Software that is able to scale horizontally could, for example, have a dynamic amount of nodes. That would allow them to use fewer resources by using fewer nodes when the resources are not necessary. They could instead be granted more nodes dynamically when the increased amount of resources is necessary.

3 Method

The prototype in this thesis, Tvilling Digital, is made from scratch. The possibility of extending the PoC from the specialization project last year was considered. It ultimately did not seem like it would be less work than starting from scratch, and it would have put a restriction on technology choices. Unknown problems in the PoC could also be problematic for the robustness of Tvilling Digital if present. As Tvilling Digital is intended to be built upon later, potentially becoming a full-fledged cloud solution for DT, the focus has been placed on making a stable and extendable base system instead of making as many features as possible. It should instead be as easy as possible to extend Tvilling Digital with new features in later projects.

3.1 Technology Research

A technology stack, the technologies to be used for the project, had to be chosen before the development of the prototype could start. The required technology stack chosen consisted of an operating systems, a programming language, a web framework, and a messaging system. A web framework was needed to create a web API that could be used for communication with the frontend, a web application created in Sande and Børhaug 2019. A messaging system was required to keep the internal flow of data standardized.

Open-source solutions have been prioritized over proprietary solutions when available to make Tvilling Digital more future-proof. If the developers of open-source solutions stop development it is still possible to maintain the solutions along with the application and continue using it. This is in contrast to proprietary solutions where even a licensing issue can be enough to prevent all usage of and development on an application until the proprietary solution is replaced. I also have a much better experience debugging open-source solutions since I am able to see what actually happens in them, unlike most proprietary solutions.

3.1.1 Operating system

The FMUs created by the version of FEDEM that was available did not include binaries for any other operating system than Windows. This, combined with the fact that only Windows versions of FEDEM was available to run simulations directly on, made other operating systems impractical for running FEDEM simulations. Since I did not want to complicate the setup and administration too much, I wanted to be able to run the entire application on the same computer when developing. Windows was consequently chosen as the operating system for the development of Tvilling Digital. Nevertheless, I decided to ensure that Tvilling Digital also supported Linux so that further development will be possible on Linux if FMUs with Linux support is used.

3.1.2 Programming language

As can be seen on Figure 4 the PoC DT solution that was created last year as part of the specialization project was made using a combination of python, javascript, and typescript. A python application was responsible for receiving data from the torsion bar suspension rig, using FEDEM to simulate the torsion bar suspension rig with the Finite Element Method (FEM) and then sending the results to a node.js application via UDP. A node.js application was written in javascript and was responsible for serving the web frontend and transmitting the data from the python application to it. The web frontend was made using a combination of javascript and typescript. This thesis focuses on the backend, namely the role of the python application and the data transmission role of the node.js application.

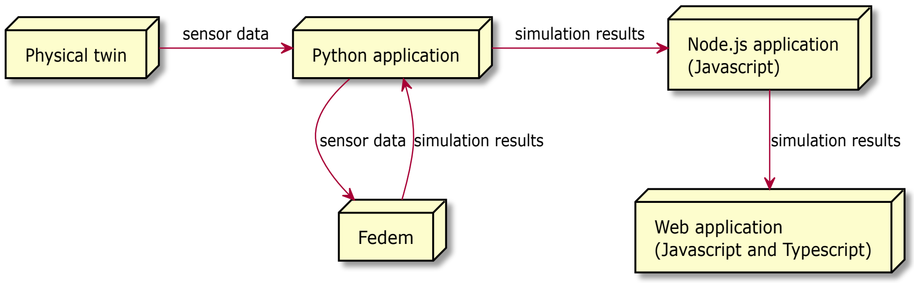


Figure 4: The components of the Proof of Concept created last year.

Python has multiple libraries available for the type of data transmission done by the node.js application in the PoC, some of which are discussed in 3.1.3. It also has an actively developed open-source library for interfacing with FMUs that is suggested by the official FMI standards page (fmi-standard 2019a). The only other languages with libraries mentioned there are C and C++. None of the other libraries listed with official Co-Simulation import support on the supported tools page[1] were viable either. Since creating a FMU library from scratch probably would have been at least a master thesis on its own I was left with Python, C or C++ as the most viable alternatives.

In addition to having a pre-existing way to interface with FMUs, Python is well known among students at NTNU. It is important that the language

[1]The official list of FMU tools on fmi-standard: `https://fmi-standard.org/tools/`

is as efficient to develop as possible, not only for this thesis but also for other theses with other students. It seems likely that future students also have more experience with Python than C or C++ since many more of the relevant courses have featured Python these last few years. Python with numpy and matplotlib is also much closer to Matlab, than C or C++ is. This is noteworthy because all the introductory programming courses at NTNU at the moment uses either Matlab or Python.

Python is also the fourth most popular language with 41.7% of developers responded that they were using it in the 2019 Stack Overflow survey, while both C and C++ have less than 25% (Stack Overflow 2019). It is also the second most loved language in the survey with 73.1% expressing interest in continuing to develop in it, much higher than the 52.0% C++ had or the 42.5% C had. Lastly, it had two web frameworks among the ten most used web frameworks in the survey, while C and C++ had none.

The fact that Python can be run dynamically and does not need to be compiled like C and C++ is also a significant benefit. Setting up and maintaining a build process has cost me much time in both C and C++ projects earlier. In my experience, I have also been able to develop faster in Python than C or C++, partially because Python is more concise with less boilerplate code.

C and C++ did have some advantages over Python, though. First and foremost, C and C++ offer better performance than python, especially when it comes to parallel execution. Python is bound by the Global Interpreter Lock (GIL)[2] which, to put it simply, means that multiple threads can't execute python code at the same time. This makes it possible for a thread to potentially freeze the entire application, which would create problems for the DTs. To prevent this, and to make use of multiple processor cores, it is necessary to create subprocesses instead. This increases the complexity of the software significantly and creates an extra performance overhead.

In spite of the problems with performance and the GIL, Python still seemed like a less complex alternative than C or C++. In addition to this, the fact that the FEDEM API was in Python and that most of the backend in the PoC from last years specialization project was in Python also contributed making Python seem like the best choice. I, therefore, decided to create Tvilling Digital in Python. The blueprints can still be developed in any language as long as they expose an interface as described in section 3.3.5 though.

[2]The Python documentation for the GIL: `https://wiki.python.org/moin/GlobalInterpreterLock`

3.1.3 Web framework

The only viable ways to transfer data to a web application with the throughput, frequency, and latency necessary for this type of application was using WebSocket, HTTP Streaming, or WebRTC. WebRTC is primarily meant for communication between web applications, and there is currently no stable python libraries supporting it. HTTP Streaming does not support sending raw binary data, only text, which would create a 33% byte overhead (Grigorik 2015, ch. 16). WebSocket, on the other hand, seemed perfect for this use-case, especially since using WebSocket worked well during the specialization project last year. There are multiple libraries supporting WebSocket available for Python. The most used libraries were aiohttp, websockets, and Django Channels.

The websockets library did not have proper support for standard HTTP interaction (Augustin 2017), and was therefore not suited for this project. Django channels has excellent support for standard HTTP interaction in addition to WebSocket and has powerful tools for database management. There was a lot of friction when trying to set it up properly and create a Minimum Viable Product (MVP) using it though, and it also required a lot of boilerplate code. Django channels was eventually abandoned after spending a significant amount of time trying to make the MVP work properly.

Aiohttp was, on the other hand, much easier to create an MVP in. There were no issues with getting it set up, and the MVP created worked without problems. Aiohttp also had support libraries for the two most popular messaging systems, Apache Kafka (through aiokafka) and Redis (through aioredis). Aiohttp was therefore chosen as the web framework for Tvilling Digital.

3.1.4 Messaging system

The platform should be able to run as multiple processes and eventually even on multiple machines at the same time. It would, therefore, be advantageous to have a structured messaging system for communication between the individual processes. Message brokers like the open-source Apache Kafka, RabbitMQ, and Redis (through the recently released Stream data type) seem ideal for the job. With a messaging broker the processes do not have to communicate with each other directly, but can instead publish to and subscribe to message queues in the broker, as seen in Figure 5. The broker will then handle the actual message transmission. This makes it considerably easier to structure Tvilling Digital, as the processes do not have to concern themselves with the communication aspect. They can simply retrieve data from the topics(Apache Kafka)/queues(RabbitMQ)/streams(Redis) they are interested in, and they can also output their results into a new topic/queue/stream as in Figure 9. There are some differences between these message brokers, though.

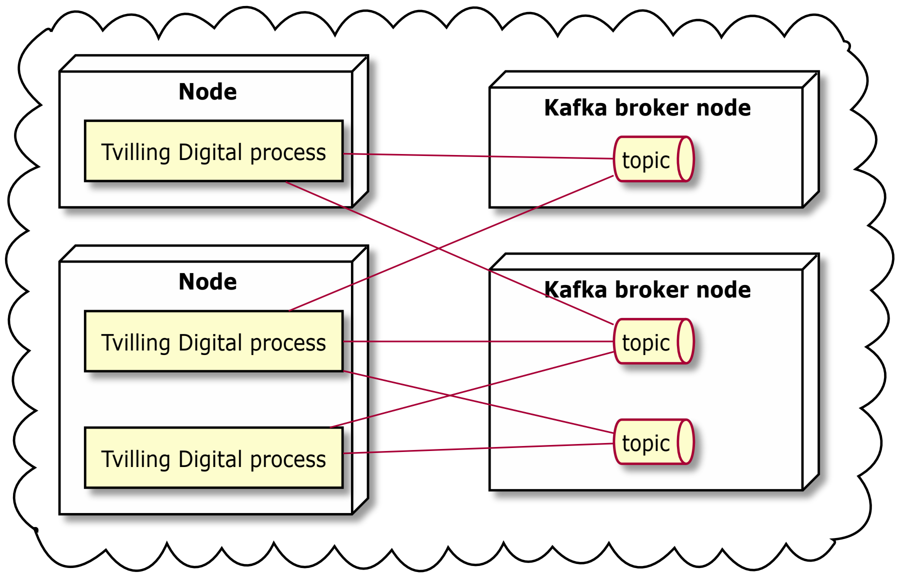


Figure 5: An example showing how Apache Kafka can be connected to multiple processes over multiple nodes/computers to enable standardized communication between them.

Redis is primarily a key-value store[3] where values of different types are stored and retrieved by keys in the form of a binary sequence. The Redis stream datatype[4] that can be used for message broking is one of many useful datatypes available, like strings, lists, or sets. Redis could potentially be used as a common data store for the processes in addition to brokering messages. The stream datatype is still quite new though, with an official release in October 2018[5], and the Redis library for aiohttp (aioredis) did not yet support it. Neither did the newest windows version of Redis, which was a problem as I had to use windows for the FEDEM FMUs. Being so new also meant that there was very little documentation and available and that it

[3]The official Redis FAQ: `https://redis.io/topics/faq`

[4]The official intro to Redis Streams: `https://redis.io/topics/streams-intro`

[5]The Redis stream release post: `https://redislabs.com/blog/redis-5-0-is-here/`

was not as battle-tested as Apache Kafka or RabbitMQ. Redis stream could be an alternative in later iterations of the system though.

Between Apache Kafka and RabbitMQ, Apache Kafka seemed like the best fit for this project. Kafka is better suited for the kind of high throughput needed for this type of system, while RabbitMQ is a more general purpose solution according to a blog post by the developers of RabbitMQ[6]. RabbitMQ also does not persist the messages, unlike Apache Kafka. There is also a support library for using Apache Kafka in aiohttp, which RabbitMQ does not have. Though Apache Kafka was chosen in this thesis, the system is created in such a way that it should not be too much work to change the messaging system later.

3.2 Prototyping tools

3.2.1 Prototyping web application client

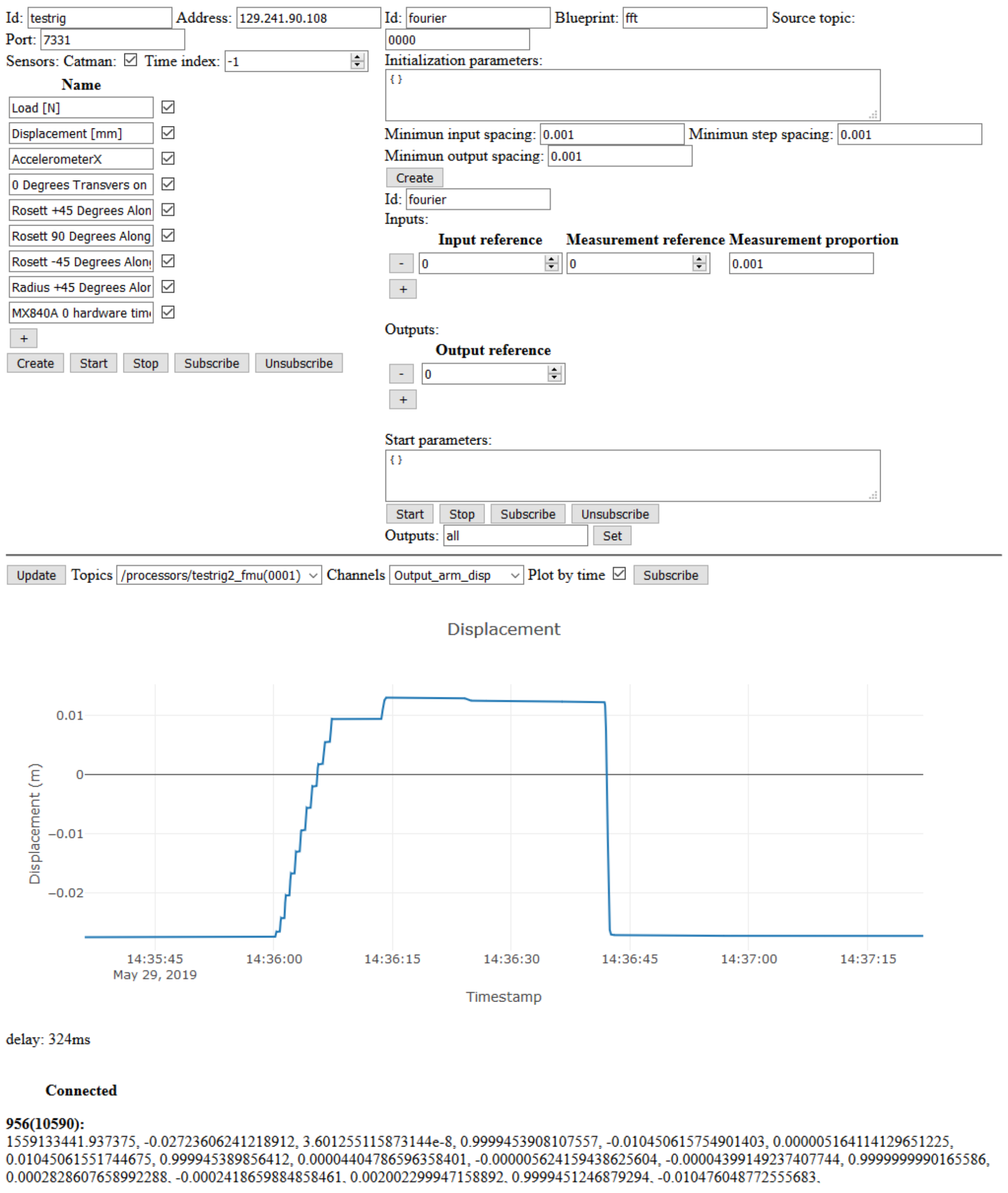
A simple web application client, shown in Figure 6, for prototyping how the API could be used by a frontend was created to ease the development. The prototyping frontend was continuously updated along with the digital twin software and was used to test changes in the software before they were made available to Sande and Børhaug 2019.

3.2.2 The torsion bar suspension rig

The torsion bar bar suspension rig, seen on Figure 7, was used for the DT prototyping. It has eight sensors:

1. Load Cell
2. Displacement
3. Accelerometer
4. 0° Strain Gauge
5. +45° Rosette
6. 90° Rosette
7. -45° Rosette
8. +45° in Radius

[6]Blog post by the developers of RabbitMQ: `https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka`



Id: testrig Address: 129.241.90.108
Port: 7331
Sensors: Catman: ☑ Time index: -1

Name	
Load [N]	☑
Displacement [mm]	☑
AccelerometerX	☑
0 Degrees Transvers on	☑
Rosett +45 Degrees Alon	☑
Rosett 90 Degrees Along	☑
Rosett -45 Degrees Alon	☑
Radius +45 Degrees Alor	☑
MX840A 0 hardware tim	☑

Id: fourier Blueprint: fft Source topic: 0000
Initialization parameters: {}
Minimun input spacing: 0.001 Minimun step spacing: 0.001
Minimun output spacing: 0.001
Id: fourier
Inputs:

Input reference	Measurement reference	Measurement proportion
0	0	0.001

Outputs:

Output reference
0

Start parameters: {}
Outputs: all

Topics /processors/testrig2_fmu(0001) Channels Output_arm_disp Plot by time ☑

delay: 324ms

Connected

956(10590):
1559133441.937375, -0.02723606241218912, 3.601255115873144e-8, 0.9999453908107557, -0.010450615754901403, 0.000005164114129651225, 0.01045061551744675, 0.999945389856412, 0.00004404786596358401, -0.000005624159438625604, -0.000043991492374077744, 0.9999999990165586, 0.0002828607658992288, -0.0002418659884858461, 0.002002299947158892, 0.9999451246879294, -0.010476048772555683,

Figure 6: Screenshot of the web application client created for testing new features before deployment

An HBM Data AcQuisition system (DAQ) is used to sample the voltage measurements from the sensors which are then sent to the installed computer via ethernet. The DAQ software Catman is used to map these voltage measurements to corresponding physical values. The results are then sent to the server as bytes using User Datagram Protocol (UDP).

An FMU was created from the rig using FEDEM. This FMU was used for most of the prototyping related to FMU simulations, both the early prototyping and when developing the blueprint-processor system and the FMU blueprint. The rig was also used for testing other processors with real-world data after testing them for correctness with the sinus and ramp processors. All of this is described in more detail in Appendix B, the report from last years specialization project, which also used the torsion bar suspension rig.

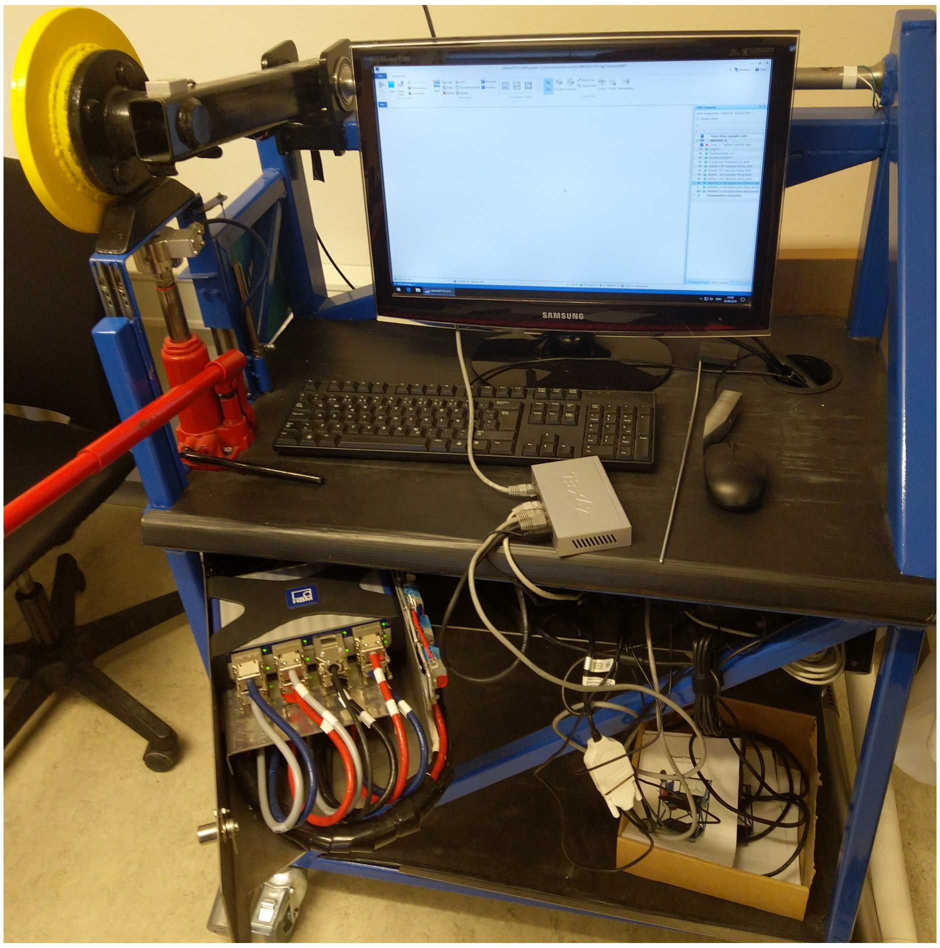


Figure 7: The torsion bar suspension rig used for prototyping.

3.3 Development

3.3.1 Configuration

Configuration of the application is done by loading attributes from a user specified file and loading them into a `Settings` object. The `Settings` object is then made available through the system. A YAML[7] file was used in early stages of development, but I eventually ended up using a python file instead. The python file can then be dynamically loaded, so that not only can python objects be put directly in the file instead of parsed, but it is also possible to dynamically generate settings values at import time as seen on Listing 1 where a new secret key is generated each time the system is run. The secret key is used for encrypting the cookies used by the API.

```python
import base64
import logging

from cryptography.fernet import Fernet

HOST = '0.0.0.0'
PORT = 1337

UDP_ADDR = ('0.0.0.0', 7331)

KAFKA_SERVER = 'localhost:9094'

FMU_DIR = 'files/fmus'
FMU_MODEL_DIR = 'files/fmu_models'
MODEL_DIR = 'files/models'
DATASOURCE_DIR = 'files/datasources'
BLUEPRINT_DIR = 'files/blueprints'
PROCESSOR_DIR = 'files/processors'

SECRET_KEY = base64.urlsafe_b64decode(Fernet.generate_key())

LOG_LEVEL = logging.INFO
```

Listing 1: An example of a settings file with dynamic generation of secret key

[7]The YAML website: `https://YAML.org/`

3.3.2 API

The API was exposed using aiohttp routes. The relevant functions were added to an aiohttp `routes` object by decorating them (a way to wrap a function in another function in Python) as in Listing 2. The routes object is then added to the applications router object which calls the function and returns the results when it receives matching http requests as can be seen in Figure 8. The real-time data is sent through WebSocket. The WebSocket connection used is made when a request is sent by a supported client to the index route of the API and kept open until the client disconnects. A client can have multiple WebSocket connections at the same time and the data the client has subscribed to will then be sent to all of them. An example of this is if the prototyping web application client mentioned in section 3.2.1 is opened in multiple browser tabs, which will result in all of them receiving the same data.

To make the API easier to use during development, it was necessary to add some form of documentation available in it. This was done by extending the aiohttp routes logic to generate an additional documentation page from the functions docstring each time a function is added to routes. This page was then reachable on the same URL as the API call, except that it had `/docs/` prepended. The documentation page for `http://tvilling.digital/processors/` would for example be available on `http://tvilling.digital/docs/processors/`.

```python
@routes.get('/processors/{id}/status', name='processor_status')
async def processor_status(request: web.Request):
  """Updates and returns the current status of the processor"""
  processor_id = request.match_info['id']
  if processor_id not in request.app['processors']:
    raise web.HTTPNotFound()
  processor_instance = request.app['processors'][processor_id]
  status = await retrieve_processor_status(
    request.app, processor_instance
  )
  return web.json_response(status, dumps=dumps)
```

Listing 2: A simple example of an API function for retrieving information.

3.3.3 Data flow

As discussed in Section 3.1.4, Apache Kafka was used for transmiting data internally in Tvilling Digital. The data that is to be transmitted from a

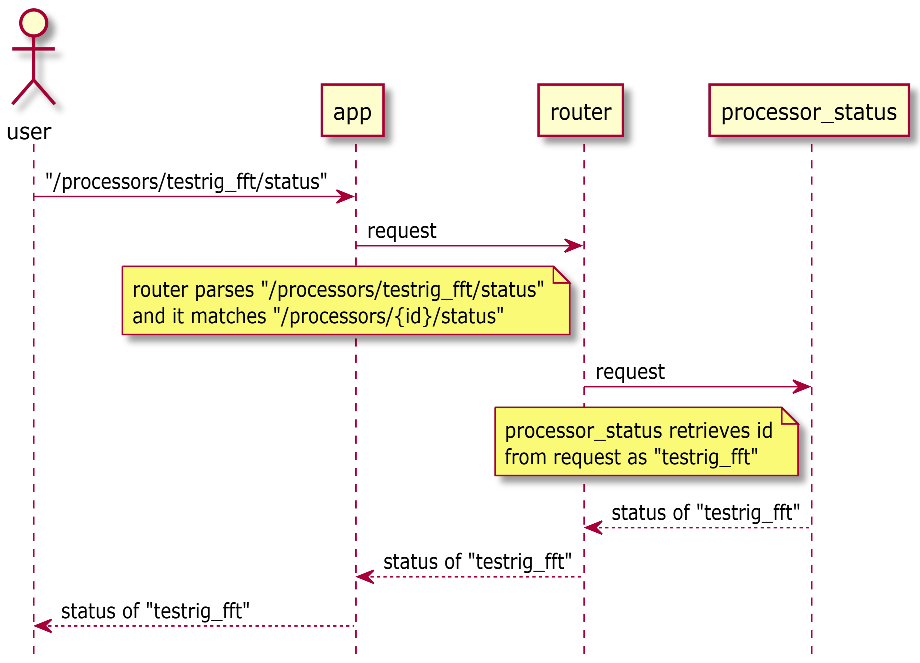

user
app
router
processor_status
"/processors/testrig_fft/status"
request
router parses "/processors/testrig_fft/status"
and it matches "/processors/{id}/status"
request
processor_status retrieves id
from request as "testrig_fft"
status of "testrig_fft"
status of "testrig_fft"
status of "testrig_fft"

Figure 8: Simplified example of request handling.

process is placed in a specific topic, which is kind of like a category, in Kafka. Other processes can then choose to receive data from that topic. Any new data added to the topic by the first process is then received by those processes. There is, in theory, no limit to the number of processes that can receive data from a topic. The API of Tvilling Digital currently allows clients to receive data from any topics with data from datasources or processors. A simplified example of data flow for a simple DT can be seen in Figure 9.

3.3.4 Datasources

The datasources in Tvilling Digital are created to make it easy receive data from multiple different sources in different formats without it affecting the other components. The datasources receive raw data from external sources and are responsible for processing it into a standardized format used internally in Tvilling Digital. Data received and processed in the datasources are then, through Kafka, made available directly in the API and as sources of data for the processors described in the following section. This separation of concerns makes it easier not only to add new types of external data sources but also to add new features that use this data, as they only have to concern themselves with one type of data format.

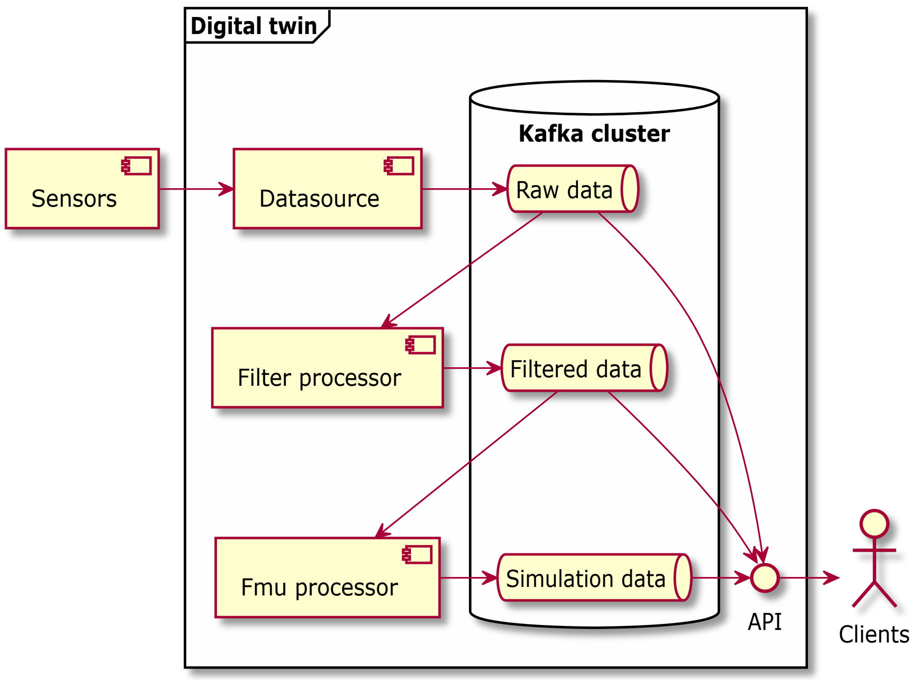


Figure 9: Simplified example of dataflow in a Digital Twin using Kafka

The processors output results are also standardized to the same format as the datasources so that there is no difference in parsing between receiving data directly from a datasource and from a processor.

3.3.5 Blueprint system definition

The blueprint system was defined to make development of the API easier by making the data processing, including the implementation of digital twins, as generic as possible. Blueprints in the blueprint system define the inner workings of the data processors. Data processors are responsible for all the data processing that form DTs in Tvilling Digital, including FE simulations, filtering, and spectral analysis. In short, they turn the data from the datasources into information the user can make use of. Processors are instantiated from blueprints in the blueprint system as shown in Figure 10.

Blueprints can in principle be any kind of algorithm, script, application, etc, but has to expose a python interface to the Tvilling Digital. The interface has to be defined in an `__init__.py` file in the root of the blueprint and has to follow some rules. Most importantly it must contain a class named P with the attributes listed below.

- They must have iterables named `input_names` and `output_names` or `inputs` and `outputs`. These must be populated after the `__init__` method is finished.
  - `input_names` and `output_names` must be a mapping from an `input_ref` or `output_ref` respectively (for example a list or tuple where the indices are the refs)
  - `inputs` and `outputs` must my an iterable where each element has a `name` attribute and a `valueReference` attribute that represents the `input_ref` or `output_ref` respectively for the name.
- The `__init__` method should initialize the processor without actually starting it. Can have an arbitrary amount of custom parameters, for example model file or buffer size. These parameters can have default values. Can change `inputs` and `outputs` or `input_names` and `output_names`.
- The `start` method is an optinal method that is called before the first time any of `set_inputs`, `get_outputs`, or `step` is called. Can have an arbitrary amount of custom parameters. These parameters can have default values.

- The `set_inputs` method should handle incoming data to the processor.
- The `get_outputs` method should return current results.
- The `step` method should perform a calculation/step in the simulation/etc.
- The `stop` method is an optional method that will be called before the processor process is stopped. Should gracefully stop the processor if necessary.

For a more practical example see listing 3

3.3.6 Blueprint system implementation

The blueprint system works by importing the `__init__.py` file dynamically in the processor process. This is done by leveraging the implementation of Pythons library import functionality, `importlib` [8]. After importing the blueprint into the processor process the P class (as seen in Listing 3) in the blueprint is instantiated and used by the processor. The P class is instantiated with arguments given by the user. These are given as a JSON object called `init_params` when creating the processor from the blueprint. The required initialization parameters will vary between blueprints though. It is, therefore, necessary to expose the required parameters to the client before importing the blueprint.

The `ast` [9] module in Python is used to extract documentations strings and function parameters from the blueprints. The `__init__.py` files describing the blueprints are parsed into Abstract Syntax Trees, By navigating through and extracting from these trees the blueprint system is able to retrieve relevant information from the blueprints without having to import them.

3.3.7 Blueprints created

Some blueprints using the blueprint system were created as part of this thesis. The created blueprints are listed below.

1. FMU

   The FMU blueprint uses the FMPy[10] library to simulate FMUs.

[8]The documentation for the Python importlib package: `https://docs.python.org/3/library/importlib`

[9]The documentation for the Python ast module: `https://docs.python.org/3/library/ast`

[10]The library used to simulate FMUs: `https://github.com/CATIA-Systems/FMPy`

```python
class P:
  # A tuple (or any iterable) with objects containing the name and
  # a reference value for each input
  inputs = (input1, input2, etc)
  # A tuple (or any iterable) with objects containing the name and
  # a reference value for each output
  outputs = (output1, output2, etc)

  def __init__(self, an_init_arg, another_init_arg='some_value'):
    """Initialize the instance using the provided args

    Can change inputs and outputs tuple
    """

  def start(self, start_time, a_start_arg, another_start_art='a_value'):
    """Start (optional)"""

  def set_inputs(self, input_refs, input_values):
    """Receive new input data"""

  def step(self, timestamp):
    """Do a calculation"""

  def get_outputs(self, output_refs):
    """Return results"""

  def stop(self):
    """Stops the blueprint instance

    An optional stop method that will be called before force-
    quitting the processor process if present.
    """
```

Listing 3: blueprint skeleton with comments

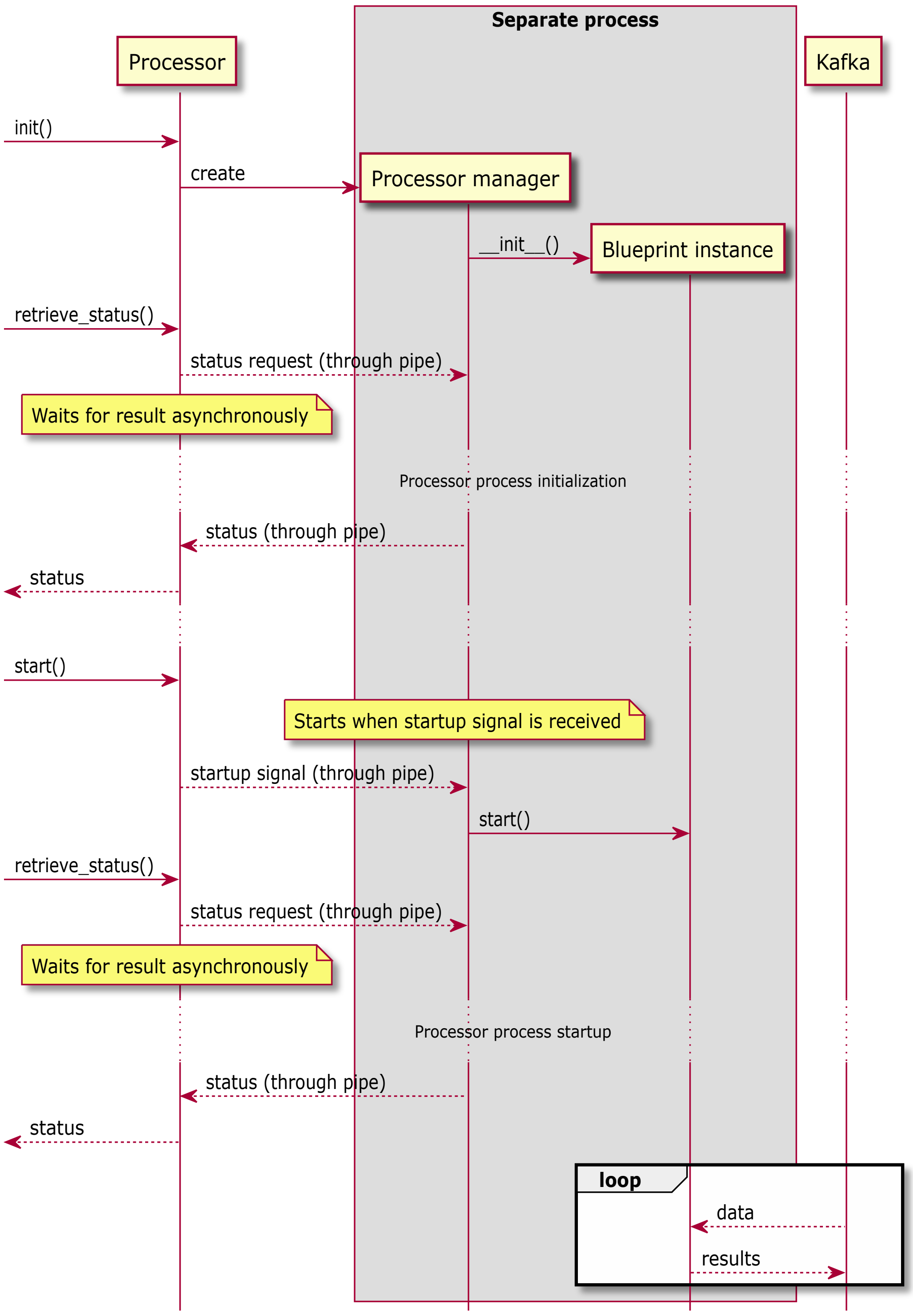

Separate process
Processor
Kafka
init()
create
Processor manager
__init__()
Blueprint instance
retrieve_status()
status request (through pipe)
Waits for result asynchronously
Processor process initialization
status (through pipe)
status
start()
Starts when startup signal is received
startup signal (through pipe)
start()
retrieve_status()
status request (through pipe)
Waits for result asynchronously
Processor process startup
status (through pipe)
status
loop
data
results

Figure 10: The startup sequence of a processor.

2. Butterworth

   The Butterworth blueprint uses the Butterworth filter in SciPy[11] to filter incoming data.

3. FFT

   The FFT blueprint uses the FFT functionality in SciPy[11] to do a spectral analysis of incoming data. It can be used for spectral analysis when the incoming data is evenly spaced.

4. Lombscargle

   The Lombscargle blueprint uses the Lombscargle functionality in SciPy[11] to do a LSSA of incoming data. It can be used for spectral analysis when the incoming data is not evenly spaced.

5. Event trigger

   The event trigger blueprint will output the given inputs only after a specified event has happened. It could be used to limit resource usage by only processing data after something relevant has happened.

6. Step

   The step blueprint will output a specified value until a specified time, and will then output another specified value. It is used to test the correctness of the other processors.

7. Sinus

   The sinus blueprint will output the sum of an arbitrary amount of specified sinus waves. It is used to test the correctness of the other processors.

3.3.8 Multiprocessing

Because Tvilling Digital had to be able to run multiple CPU intensive tasks at the same time, it was necessary to add multiprocessing functionality to it. This is done not only to utilize multiple cores but also to side-step Pythons Global Interpreter Lock (GIL) so that the processors can run in parallel without having to wait for their turn (Foundation 2019). Tvilling Digital is already asynchronous through the async/await of Python, which is necessary for aiohttp, but that is not enough to sidestep the GIL or utilize multiple

[11]The tools used for filtering and spectral analysis: `https://scipy.org/`

processor cores. This is instead achieved using the Python `multiprocessing` package [12].

After the blueprints are instantiated inside a `multiprocessing.Process` the `multiprocessing.Pipe` module is used to communicate with the process from the main thread. The messages are sent as pickled dictionaries with a `type` and a `value` item. Based on the content of `type`, which could for example be `"status"` , the contents of `value` is used in various ways. This can be seen in more detail in Figure 11. If the blueprint throws any exceptions the process will catch them, send them as a message through the pipe and then stop as can be seen in Figure 11.

3.3.9 Project structure

The files and directories of the project are structured as seen on figure 12. The source code of Tvilling Digital is placed in the `src/` folder. The source code is then organized into modules, where API endpoints are placed in `views.py` files and the code for the inner workings of Tvilling Digital is placed in `models.py` files. In addition to these there is also `kafka.py` file which is used for sending data from Apache Kafka to the API, the `utils.py` file which contains various utility code and `server.py` which is responsible for initializing Tvilling Digital. Outside of the `src/` file there is also the `main.py` file which contains the code for running the `server.py` file from the command line.

The `settings.py` file contains the configuration used when starting Tvilling Digital, `requirements.txt` contains the a list of required python libraries, and `.gitignore` is used to make git ignore certain files and directories. There is also the `html/` folder which contains the web application created for prototyping and the `docs/` which contains the documentation generation configuration and output. Lastly the `files/` folder contains all the files used and generated by the application, including FMUs, blueprints, and saved parameters. The `processors/` and `datasources/` folders contains configurations in JSON format for processors and datasources respectively. The `processors/` folder also contains the folders the processors run in and the files the processors generate (for example where the FMU contents are extracted when creating a processor from the FMU blueprint. `fmus/` contains FMUs and `fmu_models/` contains 3D models for 3D visualization of geometric FMUs. Lastly the `blueprints/` folder contains all the blueprints. Each

[12]The python multiprocessing package: `https://docs.python.org/3/library/multiprocessing`

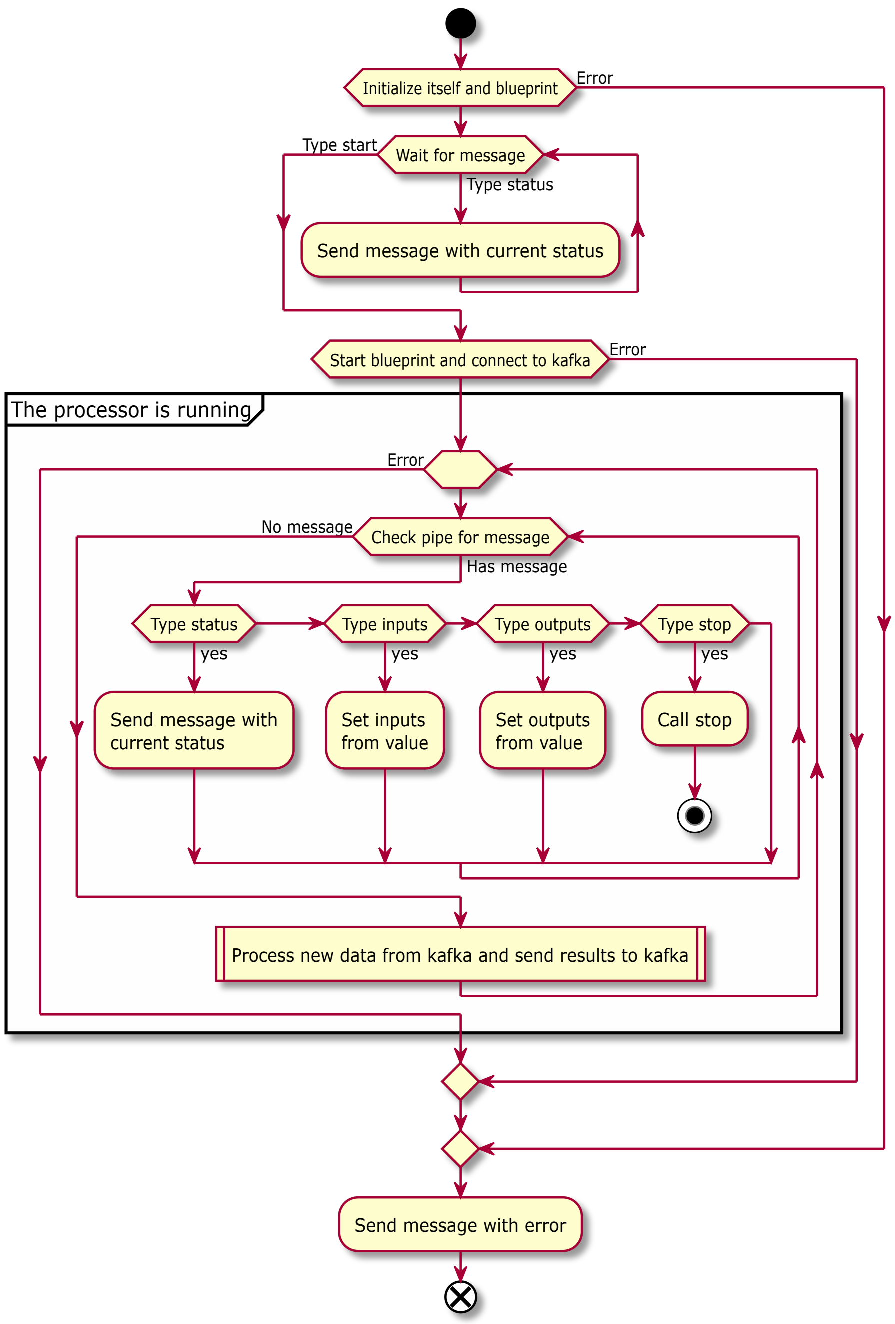


Figure 11: The lifecycle of a processor process

blueprint has its own folder with arbitrary content, except that all blueprints must have a `__init__.py` file as described in Section 3.3.5.

A heavily simplified example of how the data flows from the physical twin through the application and to the API is shown in Figure 13. Kafka has been excluded from the figure to reduce clutter. As seen on the figure, Tvilling Digital uses one datasource for each physical twin. An arbitrary amount of processors can then receive data from a datasource. An arbitrary amount of processors can again receive data from any of these processors and so on. All of these datasources and processors will also share the data directly with the API, so clients can subscribe to all data from the raw data in the datasources, through the filtered or half-processed data in the chained processors to the final results at the end of processor chains.

3.4 Documentation

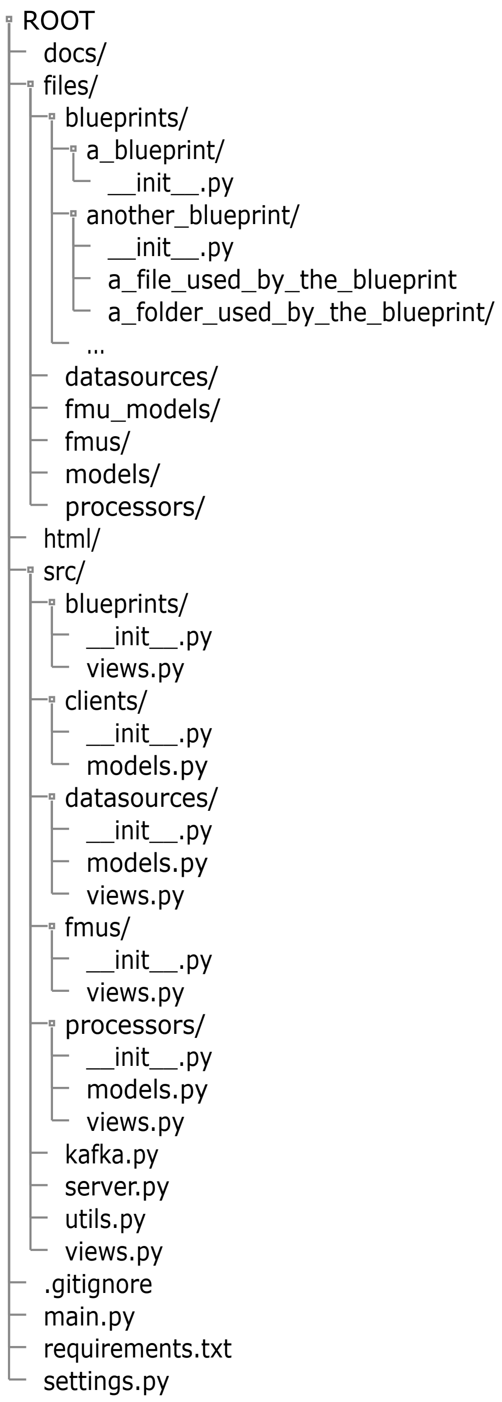
Python has built in support for writing documentation directly in the code called docstring [13]. Tvilling Digital has been heavily documented using these docstrings. This should make it much easier to reason about the various parts of the system when reading through the code. These docstrings are also used when generating the API documentation on `/docs/` and the blueprint documentation shown when retrieving blueprints in the API. Documentation in the form of a pdf, as seen in Appendix A, is also generated from these docstrings. The documentation in Appendix A is generated using the documentation tool sphinx [14]. Because the documentation is written directly in the code as docstrings with the external documentation generated from it, it should be much easier to keep the documentation updated when Tvilling Digital is further developed.

3.5 Generation of results

3.5.1 Calculation of latency

To calculate the latency of the data, the time on the DAQ on the digital twin is first synced to match the time on the computer where the latency will be tested. The latency is then calculated by comparing the timestamp of the most recent data in the browser with the current time on the computer when analyzing it in the web frontend used for testing.

[13]Python glossary: `https://docs.python.org/3/glossary.html#term-docstring`
[14]The website for the documentation tool used: `http://www.sphinx-doc.org`



```
ROOT
├ docs/
├ files/
│ ├ blueprints/
│ │ ├ a_blueprint/
│ │ │ └ __init__.py
│ │ ├ another_blueprint/
│ │ │ ├ __init__.py
│ │ │ ├ a_file_used_by_the_blueprint
│ │ │ └ a_folder_used_by_the_blueprint/
│ │ └ ...
│ ├ datasources/
│ ├ fmu_models/
│ ├ fmus/
│ ├ models/
│ └ processors/
├ html/
├ src/
│ ├ blueprints/
│ │ ├ __init__.py
│ │ └ views.py
│ ├ clients/
│ │ ├ __init__.py
│ │ └ models.py
│ ├ datasources/
│ │ ├ __init__.py
│ │ ├ models.py
│ │ └ views.py
│ ├ fmus/
│ │ ├ __init__.py
│ │ └ views.py
│ ├ processors/
│ │ ├ __init__.py
│ │ ├ models.py
│ │ └ views.py
│ ├ kafka.py
│ ├ server.py
│ ├ utils.py
│ └ views.py
├ .gitignore
├ main.py
├ requirements.txt
└ settings.py
```

Figure 12: The directory structure of the project

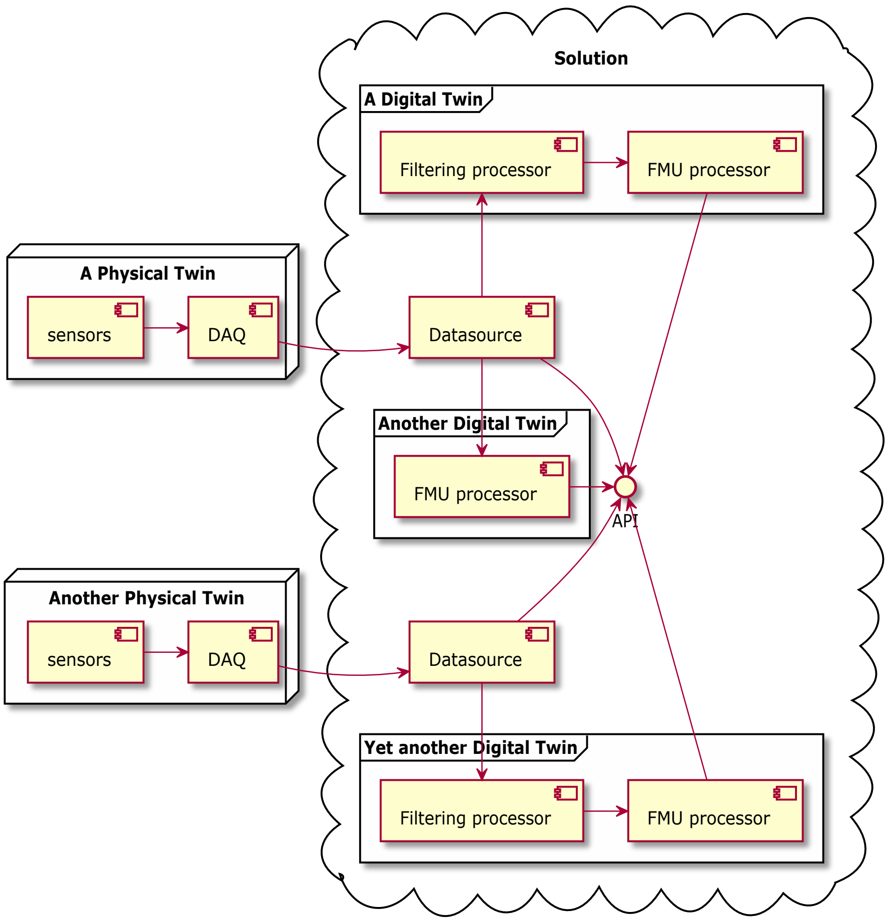


Figure 13: A heavily simplified example of how data from sensors can be distributed. The message system and other middlemen are not shown.

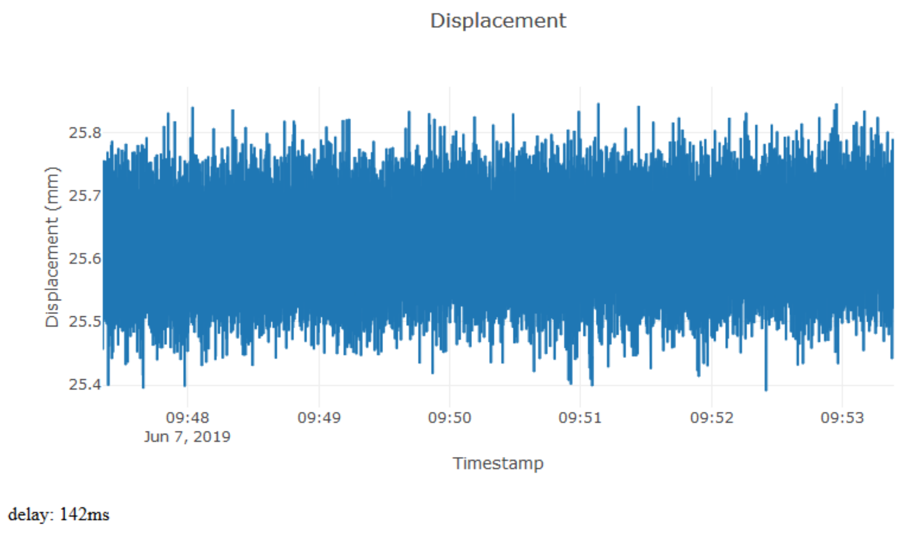
4 Results

4.1 Stability

Due to time constraints I have not been able to test the stability of the application over a longer time period than two days in a row uninterrupted. The solution did not have any stability issues or signs of memory leaks during these two days though. The processor system is made in such a way that if a processor crashes, it will not affect the rest of the system. There are nevertheless some problems with the FEDEM FMUs when running for extended periods of time. They gradually use more and more memory while simulating, which could potentially fill up the whole disk space and bring down the whole solution. This would have to be fixed in FEDEM before these FMUs could be used for indefinite amounts of time without stopping.

4.2 Latency

The latency when receiving data from a datasource object without using a processor is usually between 100ms and 200ms, as on Figure 14.



delay: 142ms

Figure 14: Example of latency when subscribing to a raw data source.

There is an additional latency of about 100ms for results data from simulations of the test rig in a FEDEM FMU using the FMU blueprint when simulating 100 steps per second as can be seen on Figure 15. There can be some latency spikes if the simulation is somehow slowed down, through for example high processor usage by another process on the computer, because the simulation will currently not skip any data. The FMU processor will then have to catch up to the newest data by simulating faster than real-time. If the FMU processor described earlier is artificially halted through pausing the process for 2-3 seconds, it requires somewhere between a half second and one second to reach below 300ms latency again.

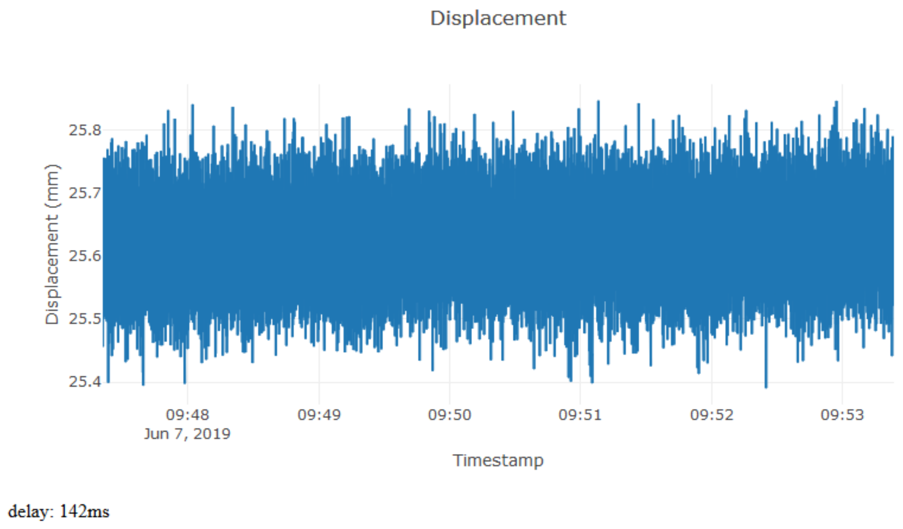


Figure 15: Example of latency when subscribing to an FMU processor

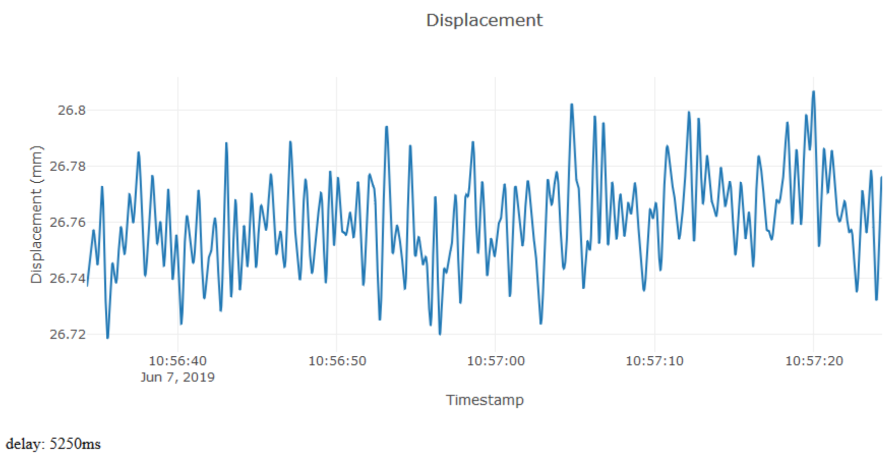
Some of the processors, like the Butterworth processor, will have an additional delay because they need to gather multiple samples before processing. They have to fill up a buffer first before they can calculate the output using the buffer. As an extreme example, if they have a buffer size of 500 measurements with a spacing of 10ms, they will have an additional 5000ms delay as can be seen in Figure 16.

4.3 Documentation

The documentation is found in the code as docstrings. An example of such a docstring can be seen in listing 4. The generated docs from listing 4 is shown in figure 17 The rest of the generated documentation is shown in Appendix A. An example of what the API documentation looks like in the browser can be seen in figure 18

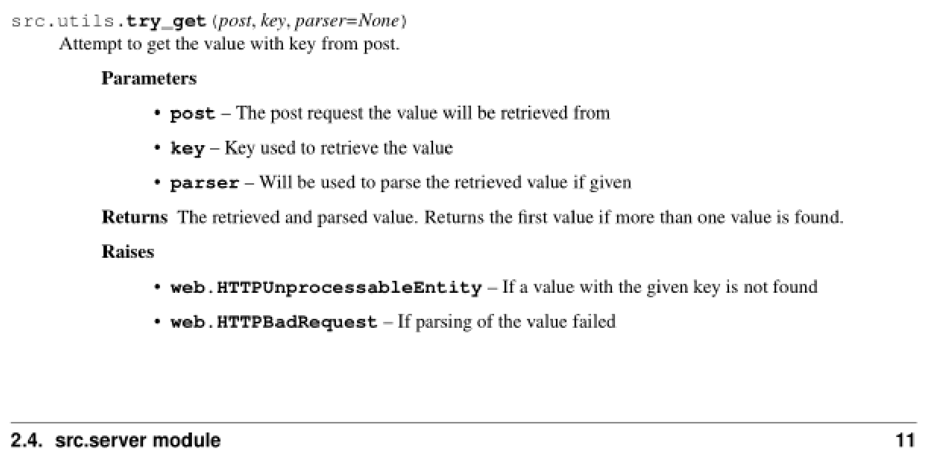
4.4 Resulting solution

A complete cloud solution with the results of this thesis as the backend and the results of Sande and Børhaug 2019 as the frontend, as shown in



delay: 5250ms

Figure 16: Example of latency when subscribing to a Butterworth processor with a buffer size of 500 measurements and a sample spacing of 10ms.



src.utils.**try_get** (*post*, *key*, *parser=None*)

Attempt to get the value with key from post.

Parameters

- **post** – The post request the value will be retrieved from
- **key** – Key used to retrieve the value
- **parser** – Will be used to parse the retrieved value if given

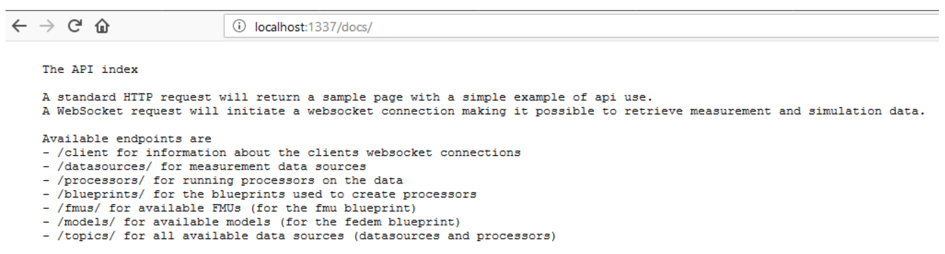
Returns The retrieved and parsed value. Returns the first value if more than one value is found.

Raises

- **web.HTTPUnprocessableEntity** – If a value with the given key is not found
- **web.HTTPBadRequest** – If parsing of the value failed

2.4. src.server module **11**

Figure 17: The generated documentation from the `try_get` function with docstring.



localhost:1337/docs/

```
The API index

A standard HTTP request will return a sample page with a simple example of api use.
A WebSocket request will initiate a websocket connection making it possible to retrieve measurement and simulation data.

Available endpoints are
- /client for information about the clients websocket connections
- /datasources/ for measurement data sources
- /processors/ for running processors on the data
- /blueprints/ for the blueprints used to create processors
- /fmus/ for available FMUs (for the fmu blueprint)
- /models/ for available models (for the fedem blueprint)
- /topics/ for all available data sources (datasources and processors)
```

Figure 18: The index page of the generated API documentsion.

```python
def try_get(post, key, parser=None):
 """
 Attempt to get the value with key from post.

 :param post: The post request the value will be retrieved from
 :param key: Key used to retrieve the value
 :param parser: Will be used to parse the retrieved value if given
 :return: The retrieved and parsed value.
          Returns the first value if more than one value is found.
 :raise web.HTTPUnprocessableEntity: If a value with the given key
                                     is not found
 :raise web.HTTPBadRequest: If parsing of the value failed
 """
 try:
  value = post[key]
  if parser:
   try:
    return parser(post[key])
   except ValueError:
    raise web.HTTPBadRequest(
     reason=f'The value {value} from {key}' +
            f' was not parsable as {parser.__name__}'
    )
  return value
 except KeyError:
  raise web.HTTPUnprocessableEntity(reason=f'{key} is missing')
```

Listing 4: The `try_get` function with docstring.

Figure 19, was created. The backend development process and the creation of the API used by the frontend is described in section 3.3. The development of a frontend using the API created in this thesis and the results of that development are described in that thesis.

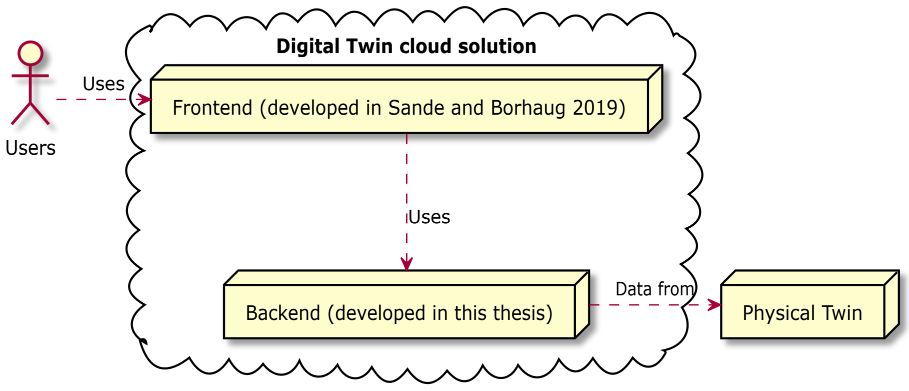

Digital Twin cloud solution
Users
Uses
Frontend (developed in Sande and Borhaug 2019)
Uses
Backend (developed in this thesis)
Data from
Physical Twin

Figure 19: An overview of the relationship between the frontend and the backend.

5 Discussion

5.1 Technology used

As discussed in section 3.1 I decided to use Windows as the operating system, Python as the programming language, aiohttp as the web framework and kafka as the messaging system. This ended up working very well, but there were some issues which could potentially be worth looking into.

Installation and setup of some of the tools turned out to be noticeably harder in Windows than on Arch Linux where they were originally tested at the start of the project. Redis, for example, was straightforward to get up and running on Arch Linux when testing messaging systems there as it was readily available from the built-in package manager. A couple of commands and a few minutes of waiting was all that was needed for the newest stable version of Redis to be installed and ready to use. The experience on Windows was very different. The newest versions of Redis available for Windows was from 2016, two years before the release of the stream datatype which was essential if Redis was to be used for this project. It could have been possible to run Redis on the Windows Subsystem for Linux, which is a compatibility layer for running Linux executables, but this has a negative impact on performance and could potentially create new issues. This was part of the reason why Kafka was chosen instead of Redis as a messaging system.

There was also a few issues with using Apache Kafka on Windows. Because Apache Kafka runs on the Java Virtual Machine it was possible to use the same binary files as other operating systems on Windows also. This made installation easy, but the startup process when using Kafka for development was unnecessarily cumbersome. Zookeeper (which Apache Kafka depends on) and Apache Kafka had to be started manually from the command line each time, in contrast to the automatic startup as a service on Linux. The most critical issue was with stability, though. This is not an issue when Tvilling Digital is deployed, because Zookeeper and Kafka do not have to run on the same computer as the rest of the system, but when developing it is much easier to run everything on the same computer. If the FEDEM FMUs later becomes available for Linux it could be worth looking into using Linux instead.

Apart from the issues known beforehand, discussed in section 3.1, there were no issues with Python or aiohttp during development.

5.2 Prototyping tools

Both the web application created for testing the API and the torsion bar suspension rig used for testing the solution with real data has been really useful. The web application made experimenting with new features and verifying that everything worked much easier than it would have been without it. The web application did become relatively messy towards the end of the project, but this was not a problem since it was only meant for personal use. Even though it was a significant amount of work to create and update it, it turned out to be worth it thanks to the boost in productivity from how easy it made testing and verification. An alternative to the web application could have been to use integration testing and maybe also unit testing instead. This could have enabled automated testing, in contrast to how changes have to be tested manually through the web application in the current version. I appreciated being able to test the changes in more practical use through the web application though. It would, in my opinion, be harder to reason about how changes would impact the users of the API when only running automated tests. An even better solution could have been to use both the web application and automated tests.

The Torsion bar suspension rig was also very useful. It made it easy to test with real data produced in real-time instead of having to generate and record data beforehand. The displacement sensor, which generated the displacement value used as input to the FMU of the torsion bar suspension rig, was somewhat inaccurate and had a high amount of noise though. A better sensor for displacement would have made correctness testing much easier. This was not a huge problem for this thesis but could be worth looking into for later projects using the torsion bar suspension rig.

5.3 Cooperation with other master students

This thesis focused on creating a backend with an API for frontends, and not the frontend itself. The frontend was instead developed in Sande and Børhaug 2019. An important part of the API development was, therefore, the communication with them. Just developing the API without accommodating their needs and simply relying on the API documentation for communicating how the API should be used would not have been sufficient. It was also necessary to cooperate with Johansen 2019, who worked on the Offshore Crane, to get an FMU of the Offshore Crane for the project.

The fact that all of us shared office space helped greatly with the communication. When features were worked on in Sande and Børhaug 2019 that utilized new API featuers, they could very quickly ask for for help, clarification on the documentation, new API featuers, or even bug fixes if a bug were found. I could then respond to them and if necessary update the documentation or API immediately, which was a huge boost to the productivity. If they for example wanted additional information included in a specific response from the server, so that they did not have to send an additional request to retrieve it separately, this could be done very quickly without forcing anyone to leave their chairs. It also allowed me to get much faster feedback on new API features and made it easy to discuss potential new changes with them before implementing the changes. It also made it easier to communicate with Johansen 2019 when working on the FMU of the Offshore Crane simulation developed there.

5.4 Results

5.4.1 Latency

The latency shown in section 4.2 seems to be acceptable for this kind of application. There may be some use-cases which require less than 100ms latency, but when it comes to structural integrity monitoring and predictive maintenance, a latency of less than a second should not be a problem. The over 5000ms of latency in Figure 16 is of as mentioned in section 4.2 only an extreme example of a possible combination of sample spacing and buffer size.

The latency and the latency variation was larger than it could have been though. This is because the data at various parts of Tvilling Digital was put in buffers and sent when the buffers reach a certain size instead of sending the data immediately. Though it negatively impacted the latency, this improved performance drastically. The CPU use of the datasources was in fact reduced with more than 90%. The latency does not have any impact on the correctness of the processors, for example the simulations, as they use the timestamp included in the data. The time used will therefore always be the time the original sensor values were measured unless a processor specifically changes it to something else.

5.5 Future work

Even though much work has been done on Tvilling Digital, there is still much more work to be done before Tvilling Digital is ready for deployment. Potential options for future work on Tvilling Digital is discussed in the following subsections.

5.5.1 Performance improvements

There is still room for performance improvements, especially when it comes to chaining multiple data processors. While Kafka itself is very performant the kafka-python library (and consequently also the aiokafka library) seems to have problems handling the number of messages desired for real-world software with a large amount of data processors. This is especially true for high-frequency data. A better solution for buffering the data before sending, or potentially even a replacement of the library could potentially improve the performance greatly. Horizontal scaling, as described in the next section, would also help by reducing the number of messages a single instance from the kafka-python library has to handle.

5.5.2 Horizontal scaling

Time restraints at the final stages of the project made it necessary to sacrifice the ability of Tvilling Digital to scale horizontally in a clean way. Restoring the horizontal scaling capability could be an important improvement to the software since there is a limit to how much software can scale vertically. The FEDEM FMUs, for example, requires a lot of resources, especially the more complex simulations and at high frequencies. Running a high number of these simulations at the same time would therefore not be feasible as long as horizontal scaling capability is missing. Horizontal scaling capability is therefore important to add for it to be feasible to serve it as a service to a large number of users at the same time.

Most of the current version of Tvilling Digital is already suitable for horizontal scaling. What remains to be done to achieve horizontal scaling capability is to replace the usage of the aiohttp `Application` instance for interaction between the API and the other parts of Tvilling Digital. The communication of data to the API is suited for horizontal scaling because of the way Apache Kafka is used, which is described in section 3.3.3. The interaction between the API and the other parts of Tvilling Digital, which is actions like starting a processor, stopping a datasource, changing the outputs of a processor, etc., could be done in a similar fashion. An external data storage could be added, and interactions would then be done by making the relevant parts of Tvilling Digital listen to changes of certain values in the storage, which could be changed through the API. This type of indirect interaction would make horizontal scaling possible.

5.5.3 Security and authorization

Security has not been in focus during the creation of Tvilling Digital and there is currently no authorization system as these are out of scope of this thesis. An authorization system and a proper security review is critical before, for example, SaaS would be possible. The system does check most of the input through the API, but as securing the system was outside of the scope of this thesis this is not guaranteed. There could, for example, be problems with the FMU processors accessing arbitrary files based on the `fmu` input parameter. The system is also vulnerable to Cross-site request forgery (CSRF). Critical actions, for example deleting processors, are done using GET instead of POST. This is not currently a problem since there is not authentication anyway, but it must be fixed if authentication is to be implemented. GET is currently used instead of POST on some critical actions because it made testing through the browser much easier. Actions could be tested by simply navigating to an URL instead of crafting a POST request through an external tool. This is easily fixed, but doing so may break current usage of the API by others.

5.5.4 Creation of more blueprints

The current blueprints, except for the FMU blueprint, are mostly made for testing the system. More and better blueprints would be an important improvement. Some useful blueprints are:

- Improved versions of existing blueprints
- A blueprint for accumulating values, which when combined with other blueprints could be used for real-time fatigue calculations
- A blueprint for using the FEDEM solver directly without an FMU. It would make testing models in the system much faster as the FMU generation usually takes a long time

There is also a lot of potential with a feature enabling users to create their own processors through the system.

It could also be worth looking into implementing these as FMUs instead of blueprints. All the work currently done by processors could potentially be done using FMUs through the FMU blueprint instead if the FMU blueprint is extended. There is also a lot of potential in implementing a feature that would enable users to create their own FMUs inside Tvilling Digital.

5.5.5 Bi-directional communication

A potential improvement to Tvilling Digital is bi-directional communication between the physical and the digital twin as described by Kritzinger et al. 2018. It would require new physical research objects that could benefit from this bi-directional communication. Windmills could, for example, automatically adjust itself (blades, generator etc.) or even shut down based on real-time data from the simulation in the digital twin.

5.5.6 Extraction of geometric models from FEDEM FMUs

Automatic extraction and conversion of geometric models from FEDEM FMUs would make it easier to add new FMUs. Currently, the geometric models have to be extracted and converted manually and placed in a directory on the server. A solution for extracting these automatically was created in Johansen 2019. This solution could potentially be integrated into Tvilling Digital.

5.5.7 Testing with FMUs using different solvers

Tvilling Digital has not yet been tested with other FMUs than FEDEM generated FMUs. While it should not be a problem to use FMUs generated by different applications, it could be beneficial to test this properly in case the FEDEM FMUs in some way deviates from the FMI standard.

5.5.8 Potential issues with FEDEM

While testing with an FMU generated from the Offshore Crane in Johansen 2019 a potential issue with the correctness of the simulation when variable timesteps were used surfaced. It seemed like the calculated forces were lower than they had been with fixed timesteps. As this was late in the project and potential issues with the FE simulation was out of the scope of this thesis, I did not investigate this beyond ensuring that it was not a problem with the data sent to the FMU or the way the results were retrieved. This potential issue could be worth looking into.

In addition to this, there was also a couple of instances where the FEDEM generated FMUs refused to start because of licensing issues with the FEDEM solver, probably because the solver was unable to reach the licensing server. Some form of solution to this would be beneficial.

5.5.9 Support for DT development

Tvilling Digital has been created with support for running DTs. This is an important first step in supporting DT development on the platform, but I would estimate that there is still multiple master's theses worth of work left. A way for users to create their own blueprints on Tvilling Digital is a potential next step. DTs would be developed as blueprints, and specialized tools for creating blueprints could then be created. One such specialized tool is a Computer-Aided Design (CAD) tool that could be used to create geometric models and choose a FE solver. A blueprint would then be created from the combination of the chosen FE solver and the geometric model. The blueprint could then be used in the current blueprint system.

5.5.10 Improvement of the blueprint system

Even though a significant amount of work has been put into the blueprint system, it is still not without possible improvements. The blueprint parser, for example, still has problems parsing default arguments that are expressions instead of simple values. This is currently not a significant problem, but it could be restricting when developing more advanced blueprints.

More advanced blueprints would also benefit from the inclusion of possible choices (on some parameters), type, parameter specific docs, etc. This can already be seen in the FMU blueprint, where the `fmu` parameter ideally would have included the possible FMUs to choose from. The API users working on the frontend (Sande and Børhaug 2019) had to retrieve these from the `/fmus/` instead, which is not an optimal solution. If the API provided the type of the parameters, it would also be possible for a frontend to provide input widgets that are better suited for the individual parameters. Input widgets could also be improved by including docs for each parameter individually. The docs for the parameters are currently a part of the method docs instead.

A way to interact with processors dynamically could also be a significant improvement to the blueprint system. The blueprint system currently only supports interaction with the inputs and outputs of a processor after startup. Being able to change parameters while the processor is running would make the blueprint system even more powerful. It would, for example, make tweaking a butterworth processor much easier by getting instant feedback when changing the order, cutoff frequency, etc. without having to restart the processor. This should be relatively easy to implement by adding, for example, a new method called `set_params` to the blueprint system and implementing it in a similar manner as the existing `start` method. It could also be possible to switch the `start` method with a similar method that is callable multiple times in a processors lifecycle, though this could make it harder for blueprints to know when they are actually started. Adding a method to the blueprint system where the blueprint could return additional status data, extending the `status` command to use it could also be helpful.

I imagine there is also a lot of other potential improvements and additions possible.

6 Summary

The goal of this master's thesis was to lay the groundwork for a platform that could serve the ability to both develop and use Digital Twins as a cloud computing resource. Relevant technologies for the creation of this platform has been investigated, and a technology stack has been defined. An extensible prototype, Tvilling Digital, was developed using the defined technology stack and a flexible system for adding filters, solvers, etc. called the blueprint system was developed. The FMU blueprint was created for running FMUs, and Tvilling Digital should in theory be able to run arbitrary DTs as long as blueprints are created for new types. Documentation both in the code, the API and in Appendix A was added. A discussion of how the prototype can be expanded on later has been done in section 5.5. There is still a significant amount of work, probably equivalent to multiple master's theses, before a full-fledged solution for both developing and running arbitrary digital twins is accomplished. The work done in this thesis should serve as a good starting point, though.

References

Augustin, Aymeric (Sept. 30, 2017). *GitHub - aaugustin/websockets: Library for building WebSocket servers and clients in Python.* URL: `https://github.com/aaugustin/websockets#why-shouldnt-i-use-websockets`.

Ayani, M., M. Ganebäck, and Amos H.C. Ng (2018). “Digital Twin: Applying emulation for machine reconditioning”. In: *Procedia CIRP* 72. 51st CIRP Conference on Manufacturing Systems, pp. 243–248. ISSN: 2212-8271. DOI: `https://doi.org/10.1016/j.procir.2018.03.139`. URL: `http://www.sciencedirect.com/science/article/pii/S2212827118302968`.

Borodulin, Kirill et al. (2017). “Towards Digital Twins Cloud Platform: Microservices and Computational Workflows to Rule a Smart Factory”. In: *Proceedings of the10th International Conference on Utility and Cloud Computing.* UCC ’17. Austin, Texas, USA: ACM, pp. 209–210. ISBN: 978-1-4503-5149-2. DOI: `10.1145/3147213.3149234`. URL: `http://doi.acm.org/10.1145/3147213.3149234`.

El Saddik, A. (Apr. 2018). “Digital Twins: The Convergence of Multimedia Technologies”. In: *IEEE MultiMedia* 25.2, pp. 87–92. ISSN: 1070-986X. DOI: `10.1109/MMUL.2018.023121167`.

fmi-standard (May 11, 2019a). *Downloads | Functional Mock-up Interface.* URL: `https://fmi-standard.org/downloads/`.

– (May 11, 2019b). *Functional Mock-up Interface.* URL: `https://fmi-standard.org/`.

Foundation, Python Software (June 2019). *multiprocessing — Process-based parallelism — Python 3.7.3 documentation.* URL: `https://docs.python.org/3/library/multiprocessing.html`.

Furht, Borivoje and Armando Escalante (2010). *Handbook of cloud computing.* Vol. 3. Springer.

Glaessgen, Edward and David Stargel (2012). “The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles”. In: *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference.* DOI: `10.2514/6.2012-1818`. eprint: `https://arc.aiaa.org/doi/pdf/10.2514/6.2012-1818`. URL: `https://arc.aiaa.org/doi/abs/10.2514/6.2012-1818`.

Grieves, Michael (2014). “Digital twin: Manufacturing excellence through virtual factory replication”. In: *White paper*, pp. 1–7.

Grigorik, Ilya (Sept. 2015). *High Performance Browser Networking.* Ed. by Melanie Yarbrough Courtney Nash. 3rd ed. O’Reilly Media, Inc. ISBN: 978-1449344764.

Haag, Sebastian and Reiner Anderl (2018). “Digital twin – Proof of concept”. In: *Manufacturing Letters* 15. Industry 4.0 and Smart Manufacturing, pp. 64–66. ISSN: 2213-8463. DOI: https://doi.org/10.1016/j.mfglet.2018.02.006. URL: http://www.sciencedirect.com/science/article/pii/S2213846318300208.

Johansen, Christian (2019). “Digital Twin of offshore knucle boom crane”. Master’s thesis. Norwegian University of Science and Technology.

Kritzinger, Werner et al. (2018). “Digital Twin in manufacturing: A categorical literature review and classification”. In: *IFAC-PapersOnLine* 51.11. 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018, pp. 1016–1022. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.08.474. URL: http://www.sciencedirect.com/science/article/pii/S2405896318316021.

Mell, Peter, Tim Grance, et al. (2011). “The NIST definition of cloud computing”. In: URL: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf.

Negri, Elisa et al. (2019). “FMU-supported simulation for CPS Digital Twin”. In: *Procedia Manufacturing* 28. 7th International conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2018), pp. 201–206. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.12.033. URL: http://www.sciencedirect.com/science/article/pii/S2351978918313763.

Padovano, Antonio et al. (2018). “A Digital Twin based Service Oriented Application for a 4.0 Knowledge Navigation in the Smart Factory”. In: *IFAC-PapersOnLine* 51.11. 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018, pp. 631–636. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.08.389. URL: http://www.sciencedirect.com/science/article/pii/S2405896318315143.

Roser, Christoph (2015). *Illustration of Industry 4.0, showing the four "industrial revolutions" with a brief English description.* Christoph Roser at AllAboutLean.com. URL: https://upload.wikimedia.org/wikipedia/commons/c/c8/Industry_4.0.png.

Sande, Odd Harald Sjursen and Andreas Børhaug (2019). “Developing a Client for a Digital Twin Cloud Platform”. Master’s thesis. Norwegian University of Science and Technology.

Schleich, Benjamin et al. (2017). “Shaping the digital twin for design and production engineering”. In: *CIRP Annals* 66.1, pp. 141–144. ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2017.04.040. URL: http://www.sciencedirect.com/science/article/pii/S0007850617300409.

Schroeder, Greyce N. et al. (2016). “Digital Twin Data Modeling with AutomationML and a Communication Methodology for Data Exchange”. In: *IFAC-PapersOnLine* 49.30. 4th IFAC Symposium on Telematics Applications TA 2016, pp. 12–17. ISSN: 2405-8963. DOI: https://doi.org/10.

1016/j.ifacol.2016.11.115. URL: http://www.sciencedirect.com/science/article/pii/S2405896316325538.

Shafto, Mike et al. (May 2010). “Modeling, Simulation, Information Technology and Processing Roadmap”. In:

Stack Overflow (2019). *Stack Overflow Developer Survey 2019.* URL: https://insights.stackoverflow.com/survey/2019.

Tao, F. et al. (Apr. 2019). “Digital Twin in Industry: State-of-the-Art”. In: *IEEE Transactions on Industrial Informatics* 15.4, pp. 2405–2415. ISSN: 1551-3203. DOI: 10.1109/TII.2018.2873186.

Uhlemann, Thomas H.-J., Christian Lehmann, and Rolf Steinhilper (2017). “The Digital Twin: Realizing the Cyber-Physical Production System for Industry 4.0”. In: *Procedia CIRP* 61. The 24th CIRP Conference on Life Cycle Engineering, pp. 335–340. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2016.11.152. URL: http://www.sciencedirect.com/science/article/pii/S2212827116313129.

A Documentation

Tvilling Digital

Simen Norderud Jensen

Jun 11, 2019

CONTENTS:

CHAPTER

ONE

MAIN MODULE

The start point of the application.

class main.**Settings**(*settings_module*)

Bases: `object`

A class for holding the application settings

main.**main**(*args*)

Start the application.

Will be called with command line args if the file is run as a script

CHAPTER

TWO

SRC PACKAGE

2.1 Subpackages

2.1.1 src.blueprints package

2.1.1.1 Submodules

2.1.1.2 src.blueprints.views module

async `src.blueprints.views.`**`blueprint_detail`**(*request: aiohttp.web_request.Request*)

Get detailed information for the blueprint with the given id

async `src.blueprints.views.`**`blueprint_list`**(*request: aiohttp.web_request.Request*)

List all uploaded blueprints.

Append a blueprint id to get more information about a listed blueprint.

`src.blueprints.views.`**`dumps`**(*obj*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=<function make_serializable>*, *sort_keys=False*, ***kw*)

A version of json.dumps that uses make serializable recursively to make objects serializable

async `src.blueprints.views.`**`retrieve_method_info`**(*class_body*, *method_name*, *params_ignore=1*) → Tuple[str, List]

Retrieves docs and parameters from the method

Parameters

- **`class_body`** – the body of the class the method belongs to
- **`method_name`** – the name of the method
- **`params_ignore`** – how many of the first params to ignore, defaults to 1 (only ignore self)

Returns a tuple containing both the docstring of the method and a list of parameters with name and default value

2.1.2 src.clients package

2.1.2.1 Submodules

2.1.2.2 src.clients.models module

class src.clients.models.**Client**

Bases: `object`

Handles connections to a clients websocket connections

async close()

Will close all the clients websocket connections

dict_repr() → dict

Returns a the number of connections the client has

async receive(*topic*, *bytes*)

Asynchronously transmit data to the clients websocket connections

Will add the data to the buffer and send it when the buffer becomes large enough

Parameters

- **topic** – the topic the data received from
- **bytes** – the data received as bytes

2.1.2.3 src.clients.views module

async src.clients.views.**client**(*request: aiohttp.web_request.Request*)

Show info about the client sending the request

src.clients.views.**dumps**(*obj*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=<function make_serializable>*, *sort_keys=False*, ***kw*)

A version of json.dumps that uses make serializable recursively to make objects serializable

2.1.3 src.datasources package

2.1.3.1 Submodules

2.1.3.2 src.datasources.models module

class src.datasources.models.**UdpDatasource**(*addr: Tuple[str, int]*, *input_byte_format: str*, *input_names: List[str]*, *output_refs: List[int]*, *time_index: int*, *topic: str = None*)

Bases: `object`

Represents a single UDP datasource

class src.datasources.models.**UdpReceiver**(*kafka_addr: str*)

Bases: `asyncio.protocols.DatagramProtocol`

Handles all UDP datasources

connection_lost(*exc: Optional[Exception]*) → None
Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

connection_made(*transport: asyncio.transports.BaseTransport*) → None
Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for data_received() calls. When the connection is closed, connection_lost() is called.

datagram_received(*raw_data: bytes, addr: Tuple[str, int]*) → None
Filters, transforms and buffers incoming packets before sending it to kafka

error_received(*exc: Exception*) → None
Called when a send or receive operation raises an OSError.

(Other than BlockingIOError or InterruptedError.)

get_sources()
Returns a list of the current sources

set_source(*source_id: str, addr: Tuple[str, int], topic: str, input_byte_format: str, input_names: List[str], output_refs: List[int], time_index: int*) → None
Creates a new datasource object and adds it to sources, overwriting if necessary

Parameters

- **source_id** – the id to use for the datasource
- **addr** – the address the datasource will send from
- **topic** – the topic the data will be put on
- **input_byte_format** – the byte_format of the data that will be received
- **input_names** – the names of the values in the data that will be received
- **output_refs** – the indices of the values that will be transmitted to the topic
- **time_index** – the index of the value that represents the time of the data

src.datasources.models.**generate_catman_outputs**(*output_names: List[str], output_refs, single: bool = False*) → Tuple[List[str], List[int], str]
Generate ouput setup for a datasource that is using the Catman software

Parameters

- **single** – true if the data from Catman is single precision (4 bytes each)
- **output_names** – a list of the names of the input data

2.1.3.3 src.datasources.views module

async src.datasources.views.**datasource_create**(*request: aiohttp.web_request.Request*)
Create a new datasource from post request.

Post parameters:

- id: the id to use for the source
- address: the address to receive data from
- port: the port to receive data from

- output_name: the names of the outputs Must be all the outputs and in the same order as in the byte stream.
- output_ref: the indexes of the outputs that will be used
- time_index: the index of the time value in the output_name list
- byte_format: the python struct format string for the data received. Must include byte order (https://docs.python.org/3/library/struct.html?highlight=struct#byte-order-size-and-alignment) Must be in the same order as name. Will not be used if catman is true.
- catman: set to true to use catman byte format byte_format is not required if set
- single: set to true if the data is single precision float Only used if catman is set to true

returns redirect to created simulation page

async `src.datasources.views.`**`datasource_delete`**(*request: aiohttp.web_request.Request*)
Delete the datasource

async `src.datasources.views.`**`datasource_detail`**(*request: aiohttp.web_request.Request*)
Information about the datasource with the given id. To delete the datasource append /delete To subscribe to the datasource append /subscribe To start the datasource append /start To stop the datasource append /stop

async `src.datasources.views.`**`datasource_list`**(*request: aiohttp.web_request.Request*)
List all datasources.

Listed datasources will contain true if currently running and false otherwise. Append an id to get more information about a listed datasource. Append /create to create a new datasource

async `src.datasources.views.`**`datasource_start`**(*request: aiohttp.web_request.Request*)
Start the datasource

async `src.datasources.views.`**`datasource_stop`**(*request: aiohttp.web_request.Request*)
Stop the server from retrieving data from the datasource with the given id.

async `src.datasources.views.`**`datasource_subscribe`**(*request: aiohttp.web_request.Request*)
Subscribe to the datasource with the given id

async `src.datasources.views.`**`datasource_unsubscribe`**(*request: aiohttp.web_request.Request*)
Unsubscribe to the datasource with the given id

`src.datasources.views.`**`dumps`**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=<function make_serializable>*, *sort_keys=False*, **kw)
A version of json.dumps that uses make serializable recursively to make objects serializable

`src.datasources.views.`**`try_get_source`**(*app*, *topic*)
Attempt to get the datasource sending to the given topic

Raises an HTTPNotFound error if not found.

2.1.4 src.fmus package

2.1.4.1 Submodules

2.1.4.2 src.fmus.views module

`src.fmus.views.`**`dumps`**`(`*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=<function make_serializable>*, *sort_keys=False*, **kw`)`
A version of json.dumps that uses make serializable recursively to make objects serializable

`async` `src.fmus.views.`**`fmu_detail`**`(`*request: aiohttp.web_request.Request*`)`
Get detailed information for the FMU with the given id

Append /models to get the 3d models if any

`async` `src.fmus.views.`**`fmu_list`**`(`*request: aiohttp.web_request.Request*`)`
List all uploaded FMUs.

Append an FMU id to get more information about a listed FMU.

`async` `src.fmus.views.`**`fmu_model`**`(`*request: aiohttp.web_request.Request*`)`
Get a 3d model belonging to the FMU if it exists

`async` `src.fmus.views.`**`fmu_models`**`(`*request: aiohttp.web_request.Request*`)`
List the 3d models belonging to the FMU if any exists

Append the models id the get a specific model

2.1.5 src.processors package

2.1.5.1 Submodules

2.1.5.2 src.processors.models module

`class` `src.processors.models.`**`Processor`**`(`*processor_id: str*, *blueprint_id: str*, *blueprint_path: str*, *init_params: dict*, *topic: str*, *source_topic: str*, *source_format: str*, *min_input_spacing: float*, *min_step_spacing: float*, *min_output_spacing: float*, *processor_root_dir: str*, *kafka_server: str*`)`
Bases: `object`

The main process endpoint for processor processes

`retrieve_status``()`
Retrieves the status of the processor process

Can only be called after initialization. Should be run in a separate thread to prevent the connection from blocking the main thread :return: the processors status as a dict

`set_inputs``(`*input_refs*, *measurement_refs*, *measurement_proportions*`)`
Sets the input values, must not be called before start

Parameters **`output_refs`** – the indices of the inputs that will be used

`set_outputs``(`*output_refs*`)`
Sets the output values, must not be called before start

Parameters

- **`input_refs`** – the indices of the inputs that will be used
- **`measurement_refs`** – the indices of the input data values that will be used. Must be in the same order as input_ref.
- **`measurement_proportions`** – list of scales to be used on values before inputting them. Must be in the same order as input_ref.

`start`(*input_refs*, *measurement_refs*, *measurement_proportions*, *output_refs*, *start_params*)

Starts the process, must not be called before init_results

Parameters

- **`input_refs`** – the indices of the inputs that will be used
- **`measurement_refs`** – the indices of the input data values that will be used. Must be in the same order as input_ref.
- **`measurement_proportions`** – list of scales to be used on values before inputting them. Must be in the same order as input_ref.
- **`output_refs`** – the indices of the inputs that will be used
- **`start_params`** – the processors start parameters as a dict

Returns the processors status as a dict

`async stop`()

Attempts to stop the process nicely, killing it otherwise

`class` `src.processors.models.`**`Variable`**(*valueReference: int*, *name: str*)

Bases: `object`

A simple container class for variable attributes

`src.processors.models.`**`processor_process`**(*connection: multiprocessing.connection.Connection*, *blueprint_path: str*, *init_params: dict*, *processor_dir: str*, *topic: str*, *source_topic: str*, *source_format: str*, *kafka_server: str*, *min_input_spacing: float*, *min_step_spacing: float*, *min_output_spacing: float*)

Runs the given blueprint as a processor

Is meant to be run in a separate process

Parameters

- **`connection`** – a connection object to communicate with the main process
- **`blueprint_path`** – the path to the blueprint folder
- **`init_params`** – the initialization parameters to the processor as a dictionary
- **`processor_dir`** – the directory the created process will run in
- **`topic`** – the topic the process will send results to
- **`source_topic`** – the topic the process will receive data from
- **`source_format`** – the byte format of the data the process will receive
- **`kafka_server`** – the address of the kafka bootstrap server the process will use
- **`min_input_spacing`** – the minimum time between each input to the processor

- **min_step_spacing** – the minimum time between each step function call on the processor
- **min_output_spacing** – the minimum time between each results retrieval from the processor

Returns

2.1.5.3 src.processors.views module

src.processors.views.**dumps**(*obj*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=<function make_serializable>*, *sort_keys=False*, ***kw*)

A version of json.dumps that uses make serializable recursively to make objects serializable

async src.processors.views.**processor_create**(*request: aiohttp.web_request.Request*)

Create a new processor from post request.

Post params:

- id:* id of new processor instance max 20 chars, first char must be alphabetic or underscore, other chars must be alphabetic, digit or underscore
- blueprint:* id of blueprint to be used max 20 chars, first char must be alphabetic or underscore, other chars must be alphabetic, digit or underscore
- init_params: the processor specific initialization variables as a json string
- topic:* topic to use as input to processor
- min_output_interval: the shortest time allowed between each output from processor in seconds

async src.processors.views.**processor_delete**(*request: aiohttp.web_request.Request*)

Delete the processor with the given id.

async src.processors.views.**processor_detail**(*request: aiohttp.web_request.Request*)

Get detailed information for the processor with the given id

Append /subscribe to subscribe to the processor Append /unsubscribe to unsubscribe to the processor Append /stop to stop the processor Append /delete to delete the processor Append /outputs to get the outputs of the processor Append /inputs to get the inputs of the processor Append /status to update and get the status of the processor

async src.processors.views.**processor_inputs_update**(*request: aiohttp.web_request.Request*)

Update the processor inputs

Post params:

- input_ref: reference values to the inputs to be used
- measurement_ref: reference values to the measurement inputs to be used for the inputs. Must be in the same order as input_ref.
- measurement_proportion: scale to be used on measurement values before inputting them. Must be in the same order as input_ref.

async src.processors.views.**processor_list**(*request: aiohttp.web_request.Request*)

List all created processors.

Returns a json object of processor id to processor status objects.

Append a processor id to get more information about a listed processor. Append /create to create a new processor instance Append /clear to delete stopped processors

async src.processors.views.**processor_outputs_update**(*request: aiohttp.web_request.Request*)

Update the processor outputs

Post params:

- output_ref: reference values to the outputs to be used

async src.processors.views.**processor_start**(*request: aiohttp.web_request.Request*)

Start a processor from post request.

Post params:

- id:* id of processor instance max 20 chars, first char must be alphabetic or underscore, other chars must be alphabetic, digit or underscore
- start_params: the processor specific start parameters as a json string
- input_ref: list of reference values to the inputs to be used
- output_ref: list of reference values to the outputs to be used
- measurement_ref: list of reference values to the measurement inputs to be used for the inputs. Must be in the same order as input_ref.
- measurement_proportion: list of scales to be used on measurement values before inputting them. Must be in the same order as input_ref.

async src.processors.views.**processor_status**(*request: aiohttp.web_request.Request*)

Updates and returns the current status of the processor

async src.processors.views.**processor_stop**(*request: aiohttp.web_request.Request*)

Stop the processor with the given id.

async src.processors.views.**processor_subscribe**(*request: aiohttp.web_request.Request*)

Subscribe to the processor with the given id

async src.processors.views.**processor_unsubscribe**(*request: aiohttp.web_request.Request*)

Unsubscribe to the processor with the given id

async src.processors.views.**processors_clear**(*request: aiohttp.web_request.Request*)

Delete data from all processors that are not running

async src.processors.views.**retrieve_processor_status**(*app*, *processor_instance*)

Retrieve the initialization results from a processor

Will put the results in app['topics'] and return them.

2.2 Submodules

2.3 src.kafka module

async src.kafka.**consume_from_kafka**(*app: aiohttp.web_app.Application*)

The function responsible for delivering data to the connected clients.

2.4 src.server module

async `src.server.`**`cleanup_background_tasks`**(*app*)
A method to be called on shutdown, closes the WebSocket, Kafka, and UDP connections

`src.server.`**`init_app`**(*settings*) → aiohttp.web_app.Application
Initializes and starts the server

async `src.server.`**`start_background_tasks`**(*app*)
A method to be called on startup, initiates the Kafka and UDP connections

2.5 src.utils module

class `src.utils.`**`RouteTableDefDocs`**
Bases: `aiohttp.web_routedef.RouteTableDef`

A custom RouteTableDef that also creates /docs pages with the docstring of the functions.

static **`get_docs_response`**(*handler*)
Creates a new function that returns the docs of the given function

`route`(*method: str, path: str, **kwargs*) → Callable[[Union[aiohttp.abc.AbstractView, Callable[[None], Awaitable[None]]]], Union[aiohttp.abc.AbstractView, Callable[[None], Awaitable[None]]]]
Adds the given function to routes, then attempts to add the docstring of the function to /docs

`src.utils.`**`dumps`**(*obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=<function make_serializable>, sort_keys=False, **kw*)
A version of json.dumps that uses make serializable recursively to make objects serializable

async `src.utils.`**`find_in_dir`**(*filename, parent_directory=''*)
Checks if the given file is present in the given directory and returns the file if found. Raises a HTTPNotFound exception otherwise

async `src.utils.`**`get_client`**(*request*)
Returns the client object belonging to the owner of the request.

`src.utils.`**`make_serializable`**(*o*)
Makes the given object JSON serializable by turning it into a structure of dicts and strings.

`src.utils.`**`try_get`**(*post, key, parser=None*)
Attempt to get the value with key from post.

Parameters

- **`post`** – The post request the value will be retrieved from
- **`key`** – Key used to retrieve the value
- **`parser`** – Will be used to parse the retrieved value if given

Returns The retrieved and parsed value. Returns the first value if more than one value is found.

Raises

- **`web.HTTPUnprocessableEntity`** – If a value with the given key is not found
- **`web.HTTPBadRequest`** – If parsing of the value failed

async `src.utils.`**`try_get_all`**(*post*, *key*, *parser=None*)
Attempt to get all values with the given key from the given post request. Attempts to parse the values using the parser if a parser is given. Raises a HTTPException if the key is not found or the parsing fails.

`src.utils.`**`try_get_topic`**(*post*)
Attempt to get the topic value from the given post request. Attempts to validate the topic value with the topic validator strings if found. Raises a HTTPException if the key is not found or the validation fails.

`src.utils.`**`try_get_validate`**(*post*, *key*)
Attempt to get the value with the given key from the given post request. Returns the first value if more than one value is found. Attempts to validate the value with the validator strings if found. Raises a HTTPException if the key is not found or the validation fails.

2.6 src.views module

async `src.views.`**`history`**(*request: aiohttp.web_request.Request*)
Get historic data from the given topic

get params: - start: the start timestamp as milliseconds since 00:00:00 Thursday, 1 January 1970 - end: (optinoal) the end timestamp as milliseconds since 00:00:00 Thursday, 1 January 1970

async `src.views.`**`index`**(*request: aiohttp.web_request.Request*)
The API index

A standard HTTP request will return a sample page with a simple example of api use. A WebSocket request will initiate a websocket connection making it possible to retrieve measurement and simulation data.

Available endpoints are - /client for information about the clients websocket connections - /datasources/ for measurement data sources - /processors/ for running processors on the data - /blueprints/ for the blueprints used to create processors - /fmus/ for available FMUs (for the fmu blueprint) - /models/ for available models (for the fedem blueprint) - /topics/ for all available data sources (datasources and processors)

async `src.views.`**`models`**(*request: aiohttp.web_request.Request*)
List available models for the fedem blueprint

async `src.views.`**`session_endpoint`**(*request: aiohttp.web_request.Request*)
Only returns a session cookie

Generates and returns a session cookie.

async `src.views.`**`subscribe`**(*request: aiohttp.web_request.Request*)
Subscribe to the given topic

async `src.views.`**`topics`**(*request: aiohttp.web_request.Request*)
Lists the available data sources for plotting or processors

Append the id of a topic to get details about only that topic Append the id of a topic and /subscribe to subscribe to a topic Append the id of a topic and /unsubscribe to unsubscribe to a topic Append the id of a topic and /history to get historic data from a topic

async `src.views.`**`topics_detail`**(*request: aiohttp.web_request.Request*)
Show a single topic

Append /subscribe to subscribe to the topic Append /unsubscribe to unsubscribe to the topic Append /history to get historic data from a topic

async `src.views.`**`unsubscribe`**(*request: aiohttp.web_request.Request*)
Unsubscribe to the given topic

2.7 Module contents

CHAPTER THREE

BLUEPRINTS PACKAGE

3.1 Submodules

3.2 blueprints.fmu module

A blueprint for running FMUs.

class `files.blueprints.fmu.`**`P`**(*fmu='testrig.fmu'*)

The interface between the application and the FMU

`start`(*start_time*, *time_step_input_ref='-1'*)

Starts the FMU

Parameters

- **`start_time`** – not used in this blueprint
- **`time_step_input_ref`** – optional value for custom time_step input

`files.blueprints.fmu.`**`prepare_outputs`**(*output_refs*)

Create FMUPy compatible value references and outputs buffer from output_refs

Parameters **`output_refs`** – list of output indices

Returns tuple with outputs buffer and value reference list

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

B Last years specialization project

Project Thesis
Cloud Software For Digital Twin Modeling And Monitoring

Christian Johansen, Simen Norderud Jensen, Andreas Børhaug,
Odd Harald Sjursen Sande, Kia Brekke

Fall 2018

Kunnskap for en bedre verden

Summary

The objective of the project is to explore and decide upon possible solutions to create a cloud-based digital twin solution with FEDEM software assisting in simulation and processing of FE models. The development of the project has been in cooperation with SAP and Ceetron, under supervision of MTP represented by Terje Rølvåg and Bjørn Haugen.

A user guide has been made to facilitate a quick-start in new environments, or as documentation together with the system overview. A prototype containing the key features required has been developed. The chosen solution is based on a local server receiving relevant data from a data acquisition system. The data is received by a server and analysed with FEDEM. The FE results are forward to a web application where motion of the asset is reproduced in a 3D model.

Contents

List of Figures

Listings

1 Introduction

The purpose of this project is to explore, test and evaluate possibilities regarding cloud based software solutions for Digital Twins using the FEDEM software for simulation and processing. The development of the project has been in cooperation with SAP and Ceetron, under supervision of MTP represented by Terje Rølvåg and Bjørn Haugen.

1.1 Background

The concept behind digital twins is to have a software replica of a physical object or process (physical twin) that can be used to better understand the system. However, the term is used loosely and its meaning varies depending on the physical twin it is representing. In this project a digital twin refers to a finite element (FE) model of a physical asset that through FE simulations, based on sensor data, can replicate the assets behaviour in real-time.

Multiple industries are looking to make use of digital twins because the development of the Internet of Things (IoT) has made sensors less expensive. The main use cases are predictive maintenance and monitoring of structural integrity. Benefits include better lifetime estimation, less need for on-site maintenance inspections and overall cost saving. To that purpose software companies are working on improving and creating new digital twin solutions to meet the demands of these industries. However, currently there are no non-proprietary digital twin solutions accessible. The Department of Mechanical and Industrial Engineering (MTP) at NTNU has a goal to develop a cloud based software solution that supports the digital twin applications both NTNU and SAP are currently developing. This project thesis lays the ground work for developing such software.

1.2 Problem Formulation

There are three main objectives in this project.

1. Write functional requirements for development of digital twin software. These should be based on hands-on experience and knowledge about technology.
2. Identify and select state-of-the-art software solutions. This includes exploration and evaluation based on usability, cost and ability to satisfy the functionality requirements.
3. Develop a prototype to test how well the requirements can be satisfied with the chosen solution.

This report will present the requirements, a system overview and a user manual on how to set up some of the parts. Furthermore, the results from prototypes developed will be displayed and explained. Finally there will be a discussion around technology options, challenges and further work.

2 Requirements

This section describes the different components needed for the digital twin cloud software, and the desired functionality that the end-user can experience.

Minimum Functionality Requirements

Physical twin
1. Measure relevant physical attributes
2. Transmit data to external server

Server
1. Receive measurement data
2. Sensor based real-time FE simulation and analysis
3. Transmit results to clients

Client
1. Be available through a browser
2. Visualise data from server in real-time
3. Save data from server to local file-system

Desired Functionality

- Real-time 2D plot of sensor data
- Real-time transformation of 3D model mirroring the physical twin
- Real-time video stream of the physical twin
- Stress analysis visualisation
- Fatigue analysis (S-N Curve)
- Possibility to save sensor values for further analysis
- Fast Fourier Transform
- Rewind in 3D visualisation and live-plot in case of interesting events

Hardware Components for Physical Twin

- Sensors
- Data Acquisition Board
- Computer(s)

3 System Overview

This section describes the system, including the physical asset, as is.

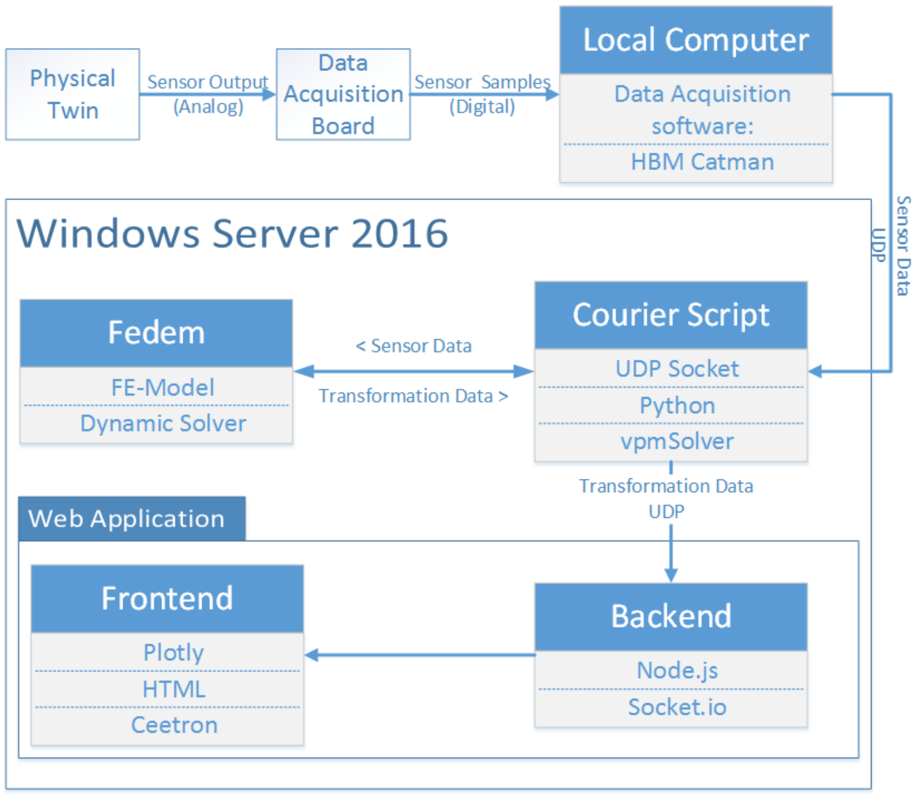


Figure 1: System overview

3.1 Physical Twin

The physical twin used in this project is the Torsion Bar Suspension Rig, which is equipped with eight sensors:

1. Load Cell
2. Displacement
3. Accelerometer
4. 0° Strain Gauge
5. +45° Rosette
6. 90° Rosette
7. −45° Rosette
8. +45° in Radius

The sensor values are sampled with an HBM data acquisition board and transferred to a computer located on the rig using an ethernet connection. More detailed information on the Torsion Bar Suspension Rig is included in appendix A.

3.2 Data Acquisition Software

The samples arriving to the computer on the Torsion Bar Suspension Rig are captured using the data acquisition software Catman. Catman is then used to map the samples values from voltage to the corresponding physical measurements. After the data is processed it is sent to the server using the remote connection option. This allows for sending data over the internet using the user datagram protocol (UDP). The remote connection option sends the data as a byte stream of 104 bytes for each time step. The mapping of the values to the bytes is shown in table 1.

Variable	**Bytes**
ID	[0:1]
Number of channels	[2:3]
Sequence counter	[4:7]
Time 1 - default sample rate	[8:15]
Time 1 - slow sample rate	[16:23]
Time 1 - fast sample rate	[24:31]
Load [N]	[32:39]
Displacement [mm]	[40:47]
AccelerometerX	[48:55]
0 Degrees Transvers on Axle	[56:63]
Rosett +45 Degrees Along Axle	[64:71]
Rosett 90 Degrees Along Axle	[72:79]
Rosett -45 Degrees Along Axle	[80:87]
Radius +45 Degrees Along Axle	[88:95]
MX840A_0 hardware time default sample rate	[96:103]

Table 1: Catman Output Format for `rigTimestamp.MEP`

3.3 Server

The server hosts the software used to represent the digital twin. For this project a virtual machine (VM) with Windows Server 2016 has been provided by NTNU IT. The following sections describe the components on the server in more detail.

3.3.1 Courier Script

The Python script (**RigSolver.py**) works as a courier between the physical twin, FEDEM and the web application. It receives sensor data from the physical twin and forwards this to FEDEM. When FEDEM is done with the dynamic analysis the results are returned and sent to the web application.

The code for **RigSolver.py** can be found in listing 1. The main functionality of the code is described below:

1. Initiate communication with the Torsion Bar Suspension Rig and the Web Application (Line 10-13)
2. Initiate communication with FEDEM solver (Line 16-20)
3. Listen for new sensor data from the Torsion Bar Suspension Rig (Line 26)
4. Unpack the sensor data to a FEDEM-friendly format (Line 30 and 33)
5. Solve dynamic analysis (Line 46)
6. Get transformation data (Line 49)
7. Send transformation data and time stamp to the web application (Line 58)

```python
import struct
import socket

from fedem.fedemdll.vpmSolverRun import VpmSolverRun

# DT setup parameters
fedem_model_path = 'TestRig.fmm'

# Configure UDP Socket
PHYSICAL_TWIN_ADDRESS = ("0.0.0.0", 7331)
WEB_SERVER_ADDRESS = ("localhost", 8001)
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(PHYSICAL_TWIN_ADDRESS)

# Initate VpmSolverRun object
with VpmSolverRun(fedem_model_path) as twin:

 # Initialization of solver (Needed for fedem functions)
 for n in range(2):
  twin.solveNext()

 # Continously receive data, solve, and forward result through
     UDP
 while(True):

  # Receive datagram
  data, _ = sock.recvfrom(32000)

  # Unpack displacement in mm from bytes 40:48 of datagram
  # Multiply with 0.001 to go from millimeters to meters
  displacement = 0.001*struct.unpack('<d',data[40:48])[0]

  # Rounding up the displacement value
  rounded_displacement=round(displacement, 4)

  # Print the sensor value
  print("Sensor value:  {} meters".format(rounded_displacement))

  # Get current time. Needed for Fedem
  time = twin.getCurrentTime()

  # Connects sensor input to correct channel (Model spesific)
  # Set extfunc channel '2' as time 'time' with data '
      rounded_displacement'.
  twin.setExtFunc(1, time, rounded_displacement)

  # Solves dynamic analysis for this time step based on sensor
      input
  twin.solveNext()
```

```

    # Get transformation data for all triads and parts
    transformationData = twin.save_transformation_state()

    # Retrieve timestamp from received datagram
    timestamp = data[96:104]

    # Assemble message with timestamp and transformationData
    message = timestamp + transformationData

    # Sends timestamp and transformation data to web client
    sock.sendto(message, WEB_SERVER_ADDRESS)
```
Listing 1: RigSolver.py

3.3.2 FEDEM

FEDEM is used to run dynamic analysis on the FE-model of the physical twin. The analysis is based on the sensor input from the physical twin and outputs transformation data for the triads and parts in the model. This is made possible by the external functions option in FEDEM. The output is an array containing the data type *double*. The format of the output array is shown in table 2.

Variable	**Element**
Time step	[0]
Time	[1]
Step length	[2]
Triad/Part	[3:17]
Triad/Part	[18:32]
⋮	⋮
Triad/Part	[End-14:End]

Table 2: Transformation Data array

Each sub array "Triad/Part" is on the format shown in table 3. "Object-Type" equals "1" for triads and "2" for parts.

Variable	**Element**
ObjectType	[0]
BaseID	[1]
Rotation Matrix $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$	[2:10]
Translation vector $\begin{bmatrix} 11 \\ 12 \\ 13 \end{bmatrix}$	[11:13]

Table 3: Triad/Part Transformation Data array

FEDEM is also used to create the surface model used for 3D visualisation from a volume model of the Torsion Bar Suspension Rig.

3.3.3 Web Application

The `Node.js` script **index.js** (Listing 2) receives the transformation data from **RigSolver.py** (Listing 1) and parses it. It uses `socket.io` to send the relevant parsed data via WebSockets to **index.html** (Listing 3).

The code for **index.js** can be found in listing 2. The main functionality of the code is described below:

1. Initialise HTTP server (Line 10-12 and 17-20)
2. Serve files required by visualisation module (Line 14-15)
3. Parse and forward incoming data (Line 32-60)
4. Listen for new data (Line 63)

```
 1 // Import and initialise libraries
 2 const express = require('express');
 3 const app = express();
 4 const http = require('http').Server(app);
 5 const io = require('socket.io')(http);
 6 const dgram = require('dgram');
 7 const struct = require('python-struct');
 8
 9 // Serve index.html when users visits the page
10 app.get('/', function(req, res) {
11     res.sendFile(__dirname + '/index.html');
12 });
13
14 app.use('/ceetron', express.static('ceetron'));
15 app.use('/js', express.static('js'));
16
17 // Start the http server for serving index.html
18 http.listen(1337, function(){
19     console.log('listening␣on␣*:1337');
20 });
21
22 // Create socket listening for new data from solver
23 fedemSocket = dgram.createSocket('udp4');
24
25 // Print to console when ready to listen for new data
26 fedemSocket.on('listening', function(){
27     const address = fedemSocket.address();
28     console.log('listening␣on␣' + address.address + ':' +
           address.port);
29 });
```

```

// Function for parsing new data from solver
fedemSocket.on('message', function(message, remote){

    // Extract timestamp from message
    const timestamp = struct.unpack('<d', message)[0]*1000;

    // Iterate over the bytes from the solver
    // Skip the timestamp and the first 3 doubles (24 bytes)
    // The bytes represents an array of doubles with a size of 8 bytes each
    // Iterate over the remaining doubles, 14 doubles at a time (112 bytes)
    for (var i = 32; i < message.length-111; i += 112) {
        // Read the baseId as the second of the 14 doubles
        const baseId = struct.unpack('<d', message.slice(i+8));
        if (baseId[0] === 318) {
            // Read the vertical displacement of the element from the message
            const displacement = struct.unpack('<d', message.slice(i + 96))[0];
            // Send the vertical displacement to the client
            io.emit('new_data', [timestamp, displacement]);
        } else if (baseId[0] === 316) {
            const t = struct.unpack('<12d', message.slice(i+16));
            const m = [
                t[0], t[1],  t[2],  0,
                t[3], t[4],  t[5],  0,
                t[6], t[7],  t[8],  0,
                t[9], t[10], t[11], 1
            ];
            io.emit('transformation', m);
        }
    }
});

// Start listening for new data from solver
fedemSocket.bind(8001, '0.0.0.0');
```

Listing 2: index.js

index.html (Listing 3) is used to plot the sensor data in the browser client and display a 3D model of the torsion bar suspension rig that replicates the movement of the asset. `Socket.io` is used to receive data sent by **index.js** (listing 2), `Ceetron Cloud Components` is used for 3D graphics and `plotly` is used for plotting.

The code for **index.html** can be found in listing 3. The main functionality of the code is described below:

1. Add toolbox for configuring 3D model draw style (Line 11-23)
2. Initialise connection to HTTP server (Line 34)
3. Initialise 3D visualisation from **usg.ts** (Listing 4) (Line 37-76)
4. Initialise plot (Line 85-98)
5. Plot live datastream (Line 103-115)
6. Update 3D model (Line 17-124)
7. Add save functionality (Line 126-152)

```
 1<!doctype html>
 2<html lang="en">
 3<head>
 4    <title>Digital Twin</title>
 5    <link rel="style.css">
 6    <script src="/socket.io/socket.io.js"></script>
 7    <script src="https://cdn.plot.ly/plotly-latest.js" charset="
        utf-8"></script>
 8</head>
 9<body style="margin:␣0;␣height:100vh;␣display:␣grid;␣grid:␣
    minmax(400px,␣50%)␣minmax(200px,␣50%)␣/␣minmax(400px,␣100%)">
10
11<div style="display:␣flex">
12    <div id="chartContainer" style="width:␣100%"></div>
13    <div style="display:␣flex;␣flex-direction:␣column">
14        <button onclick="save()">Save</button>
15        <div style="flex-grow:␣1"></div>
16        <span>Model Style</span>
17        <button onclick="myApp.setDrawStyle('surface')">Surface<
            /button>
18        <button onclick="myApp.setDrawStyle('surface_mesh')">
            Surface Mesh</button>
19        <button onclick="myApp.setDrawStyle('outline_mesh')">
            Outline Mesh</button>
        <button onclick="myApp.setDrawStyle('lines')">Lines</
            button>
        <button onclick="myApp.setDrawStyle('points')">Points</
            button>
        <button onclick="myApp.setDrawStyle('outline')">Outline<
            /button>
    </div>
</div>

<div style="line-height:␣0">
    <canvas id="CeetronCanvas"></canvas>
</div>

<script src="ceetron/require.js"></script>
<script>

    // Initialise connection to server
    var socket = io();

    // Initialise USG module
    var myApp = null;
    require(["js/usg"], function(appModule) {
        myApp = appModule.startApp("CeetronCanvas");

        // Retrieve arm geometry
        var oReq = new XMLHttpRequest();
        oReq.onload = armLoaded;
        oReq.open("get", "/js/arm.json", true);
        oReq.send();
    });

    function armLoaded(e) {
        // Send arm geometry to visualiser
        data = JSON.parse(this.responseText);
        myApp.addArmGeometry(data);

        // Retrieve torsion rod geometry
        var oReq = new XMLHttpRequest();
        oReq.onload = rodLoaded;
        oReq.open("get", "/js/TorsionRod.json", true);
        oReq.send();
    }

    function rodLoaded(e) {
        // Send torsion rod geometry to visualiser
        data = JSON.parse(this.responseText);
        myApp.addRodGeometry(data);

        // Retrieve frame geometry
        var oReq = new XMLHttpRequest();
        oReq.onload = frameLoaded;
        oReq.open("get", "/js/Frame.json", true);
        oReq.send();
    }

    function frameLoaded(e) {
        // Send frame geometry to visualiser
        data = JSON.parse(this.responseText);
        myApp.addFrameGeometry(data);
    }

    // Store reference to container for plot
    var graphContainer = document.getElementById('chartContainer
        ');

    // Container for displacement plot data
    var displacements = {x:[[]], y:[[]]};

    // Initialise plot
    Plotly.newPlot(
        graphContainer,
        [{y:[]}],
        {
            title: 'Displacement',
            xaxis: {
                title: 'Displacement (mm)'
            },
            yaxis: {
                title: 'Timestamp'
            }
        },
        {responsive: true}
    );

    // Counter for how many data points has been received
    var dataRecievedCount = 0;

    // Update plot with new data for every 100 new data points
    socket.on('new data', function(msg){
        displacements.x[0].push(new Date(msg[0]));
        displacements.y[0].push(msg[1]);
        // If 100 data points recieved since last update
        if (dataRecievedCount++ % 100 === 0) {
            // Remove points received more than 1000 points ago
            displacements.x[0] = displacements.x[0].slice
                (-100000);
            displacements.y[0] = displacements.y[0].slice
                (-100000);
            // Update plot
            Plotly.restyle(graphContainer, displacements);
        }
    });

    // Update transformation of model for every 100 new data points
    socket.on('transformation', function(msg){
        if (dataRecievedCount % 100 === 0) {
            if (myApp !== null) {
                myApp.updateDisplacement(msg);
            }
        }
    });

    // Create download dialog for currently plotted data
    function save() {
        var saveData = "Timestamp,␣displacement(mm)\r\n";
        for (var i = 0; i < displacements.x[0].length; i++) {
            saveData += displacements.x[0][i].valueOf() + ",␣" + displacements.y[0][i] + "\r\n"
        }
        download(saveData, "twin_" + new Date().toISOString() + ".csv", "text/csv");
    }

    // Downloading data to a file
    function download(data, filename, type) {
        var file = new Blob([data], {type: type});
        if (window.navigator.msSaveOrOpenBlob) // IE10+
            window.navigator.msSaveOrOpenBlob(file, filename);
        else { // Others
            var a = document.createElement("a"),
                url = URL.createObjectURL(file);
            a.href = url;
            a.download = filename;
            document.body.appendChild(a);
            a.click();
            setTimeout(function() {
                document.body.removeChild(a);
                window.URL.revokeObjectURL(url);
            }, 0);
        }
    }
</script>
</body>
</html>
```

Listing 3: index.html

The module **usg.ts** (Listing 4) is used to visualise movement in the torsion bar suspension rig through a 3D model. The model is translated and rotated according to the transformation given in the update method (which is calculated in the FEDEM solver). The drawing style of the geometry is changed through the setDrawStyle method.

The code for **usg.ts** can be found in listing 4. The main functionality of the code is described below:

1. Import Ceetron USG module used for creating, transforming and displaying the geometry. (Line 1)
2. Initialisation (Line 4-10)
3. Define class used to handle the visualisation (Line 13-141)
4. Initialise the visualisation state (Line 16-50)
5. Create the geometry representing the torsion arm (Line 65-70)
6. Create the geometry representing the torsion rod (Line 72-86)
7. Create the geometry representing the frame (Line 88-94)
8. Display statistics about the geometry in bottom left corner (Line 96-111)
9. Update arm geometry according to transformation (from FEDEM) (Line 113-127)
10. Change the drawing style of the visualisation (Line 130-141)

```
import * as cee from "../ceetron/CeeCloudClientComponent";

// Initialiser for Ceetron module of application
export function startApp(canvasElementId: string): App {
    let canvas = document.getElementById(canvasElementId);
    if (!(canvas instanceof HTMLCanvasElement)) {
        throw("Could not get canvas element");
    }
    return new App(canvas);
}

// Class containing Ceetron Cloud Client Component state
export class App {

    // Ceetron Cloud Client Component state
    private cloudSession: cee.CloudSession;
    private view: cee.View;
    private model: cee.usg.UnstructGridModel;
    private state: cee.usg.State;

    // Canvas containing visualisation
    private canvas: HTMLCanvasElement;

    constructor(canvas: HTMLCanvasElement) {
        this.canvas = canvas;

        // Initialise Ceetron Cloud Client Component
        this.cloudSession = new cee.CloudSession();
        let viewer = this.cloudSession.addViewer(canvas);
        if (!viewer) {
            throw("No WebGL support");
        }
        this.view = viewer.addView();
        this.model = new cee.usg.UnstructGridModel();
        this.view.addModel(this.model);
        this.state = this.model.addState();
        this.state.geometry = new cee.usg.Geometry();

        // Hide infoBox initially
        this.view.overlay.infoBoxVisible = false;

        // Listen for resize events
        window.addEventListener('resize', () => this.
            _handleWindowResizeEvent());

        // Manually run resize function once
        this._handleWindowResizeEvent();

        // Update view every browser frame
        window.requestAnimationFrame((time: number) => this._myAnimationFrameCallback(time));
    }

    // Adjust view dimension (called when window is resized)
    private _handleWindowResizeEvent() {
        let canvasWidth = window.innerWidth;
        let canvasHeight = this.canvas.parentElement.offsetHeight;
        this.cloudSession.getViewerAt(0).resizeViewer(canvasWidth, canvasHeight);
    }

    // Update view (called every browser frame)
    private _myAnimationFrameCallback(highResTimestamp:number) {
        this.cloudSession.handleAnimationFrameCallback(highResTimestamp);
        window.requestAnimationFrame((time: number) => this._myAnimationFrameCallback(time));
    }

    // Create the torsion arm geometry
    addArmGeometry(data) {
        let geometry = this.state.geometry.addPart();
        geometry.mesh = new cee.usg.Mesh(data.nodeArr, data.elementTypeArr, data.elementNodeIndexArr);
        geometry.settings.color = new cee.Color3(.1,.1,.1);
    }

    // Create the torsion rod geometry
    addRodGeometry(data) {
        let geometry = this.state.geometry.addPart();
        geometry.mesh = new cee.usg.Mesh(data.nodeArr, data.elementTypeArr, data.elementNodeIndexArr);
        geometry.settings.color = new cee.Color3(.8, .8, .8);

        // Transform to global coordinate system
        const c = cee.Mat4.fromElements(
            1, 0, 0, -0.02407066,
            0, 1, 0, -0.02722985,
            0, 0, 1, 0.27199998,
            0, 0, 0, 1
        );
        this.state.setPartTransformationAt(1, c);
    }

    // Create the frame geometry
    addFrameGeometry(data) {
        let geometry = this.state.geometry.addPart();
        geometry.mesh = new cee.usg.Mesh(data.nodeArr, data.elementTypeArr, data.elementNodeIndexArr);
        geometry.settings.color = new cee.Color3(.2, .2, .7);
        this.showStatistics(this.state.geometry);
    }

    private showStatistics(geometry) {
        // Generate statistics on geometry
        let nodeCount = 0;
        let elementCount = 0;
        for (let part of geometry.getPartArray()) {
            nodeCount += part.mesh.nodeCount;
            elementCount += part.mesh.elementCount;
        }

        // Log generated statistics
        console.log("Initial␣state␣loaded,␣nodeCount=" + nodeCount + ",␣elementCount=" + elementCount);

        // Draw generated statistics in bottom right corner
        this.view.overlay.infoBoxVisible = true;
        this.view.overlay.setInfoBoxContent(`Elements: ${elementCount} elements \nNodes: ${nodeCount} nodes`);
    }

    updateDisplacement(transformationMatrix: number[]) {
        // Create Ceetron matrix from transformation data
        const m = cee.Mat4.fromArray(transformationMatrix);

        const localToGlobalTransformation = cee.Mat4.fromElements(
            1, 0, 0, -0.00000001,
            0, 1, 0, -0.00000000,
            0, 0, 1, 0.00199997,
            0, 0, 0, 1
        );
        const transformation = cee.Mat4.multiply(m, localToGlobalTransformation);

        // Apply transformation to armGeometry
        this.state.setPartTransformationAt(0, transformation);
    }

    // Change drawing style for geometries
    setDrawStyle(ds: string) {
        const geometry = this.model.getStateAt(0).geometry;
        for (let part of geometry.getPartArray()) {
            if      (ds === "surface")              part.settings.drawStyle = cee.usg.DrawStyle.SURFACE;
            else if (ds === "surface_mesh")         part.
                settings.drawStyle = cee.usg.DrawStyle.
                SURFACE_MESH;
            else if (ds === "outline_mesh")         part.
                settings.drawStyle = cee.usg.DrawStyle.
                SURFACE_OUTLINE_MESH;
            else if (ds === "lines")                part.
                settings.drawStyle = cee.usg.DrawStyle.LINES;
            else if (ds === "points")               part.
                settings.drawStyle = cee.usg.DrawStyle.POINTS;
            else if (ds === "outline")              part.
                settings.drawStyle = cee.usg.DrawStyle.OUTLINE;
        }
    }
}
```

Listing 4: usg.ts

4 Web Application Prototype

The web application prototype is available at
`http://tvilling.digital:1337` when connected to the NTNU network. Figure 2 shows a digital representation of the physical asset.

The upper half of the web browser consists of a live 2D plot of the torsion arm displacement. Extra functionality for the plot window such as zoom and pan can be found in the toolbox at the top-right of the plotting window. To the right of the toolbox there is a save button. By pressing this button you can download a CSV-file to your own computer containing the previous 100 000 data points and their associated timestamp. The timestamp is saved using the Unix time standard, which is number of seconds elapsed since 1st of January 1970. A visualisation of the torsion bar suspension rig is shown on the bottom half. A model of the torsion bar suspension rig moves according to the movement of the torsion arm calculated in FEDEM. It is possible to change the zoom and camera position by scrolling and dragging, and the draw style can be changed with the buttons above on the right.

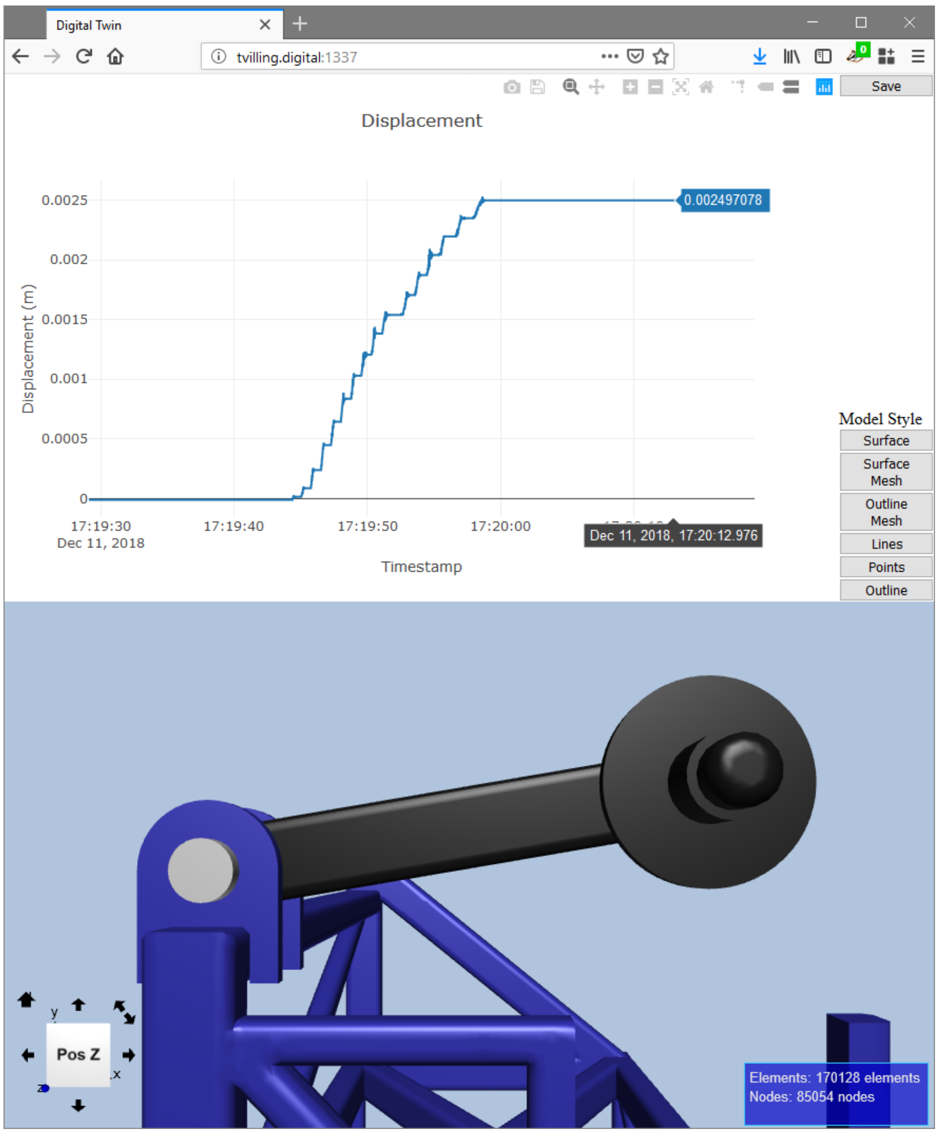


Figure 2: Digital Twin

5 User guide

This section is a user guide on how to setup the digital twin cloud software with the Torsion Bar Suspension Rig. Each subsection describes a part of the system and how to configure it.

5.1 Ethernet

The computer on the Torsion Bar Suspension Rig needs to be connected to the data acquisition board and with the WIN.NTNU.NO network through a common ethernet connection. This can be achieved by using an ethernet switch. On the Torsion Bar Suspension Rig the ethernet connection is already set up.

5.2 Catman configuration for Torsion Bar Suspension Rig

NOTE:

- The username and password for the computer is written on top of it
- Catman must be run in Administrator mode for the remote connection to work properly

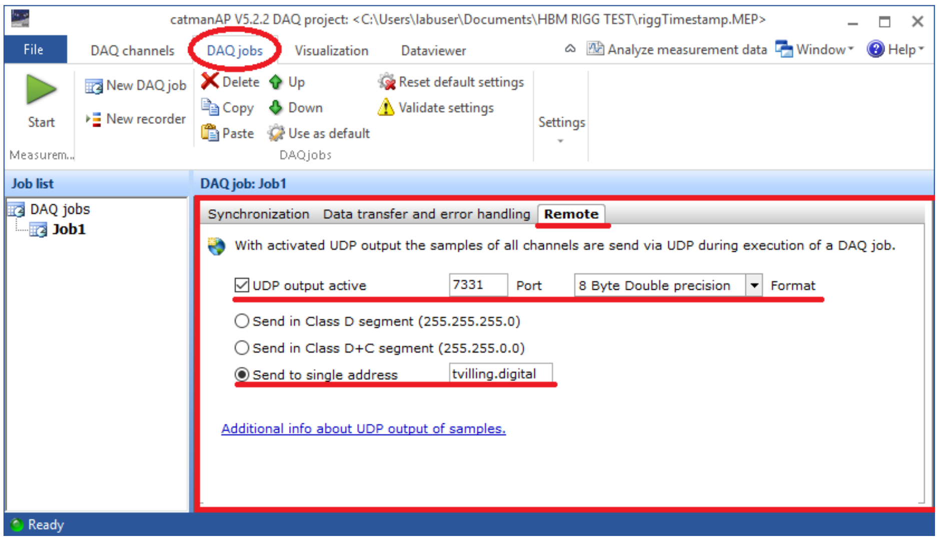
5.2.1 Initialisation and Calibration

First navigate to the directory: *C:\Users\labuser\Documents \HBM RIGG TEST* and run the file `riggTimestamp.MEP`. This will open Catman with the correct setup. Next you need to calibrate the sensors. The calibration procedure is explained in the Torsion Bar Suspension Rig Manual found in appendix A.
NOTE: This manual is designed for the project file `RIGGOPPSETT.MEP` and some of the functionality it describes is not available for the project file `riggTimestamp.MEP`.

5.2.2 Remote Connection

To set up the remote connection to the server you need to:
Go to **DAQJobs** in the header > Choose **Advanced** and then **Remote**.
The window should look like figure 3.
In this window you need to:

- Check the option for *UDP output active*
- Fill in the server port number (7331)
- Choose the format *8 Byte Single precision*
- Choose *Send to single address* and fill in the IP-address of the server (tvilling.digital or 10.212.25.104)


catmanAP V5.2.2 DAQ project: <C:\Users\labuser\Documents\HBM RIGG TEST\riggTimestamp.MEP>

DAQ job: Job1

Synchronization | Data transfer and error handling | Remote

With activated UDP output the samples of all channels are send via UDP during execution of a DAQ job.

[✓] UDP output active — Port: 7331 — Format: 8 Byte Double precision

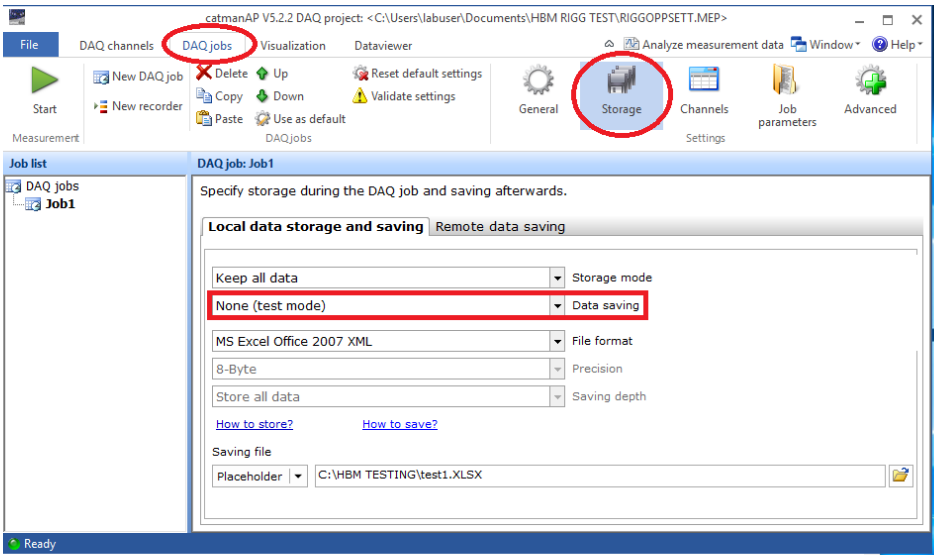
( ) Send in Class D segment (255.255.255.0)
( ) Send in Class D+C segment (255.255.0.0)
(●) Send to single address: tvilling.digital

Additional info about UDP output of samples.

Figure 3: Remote Connection in Catman

5.2.3 Storage

If not specified, Catman will locally store all data recorded. To avoid this: Go to **DAQJobs** in the header > Choose **Storage** and then **Local data storage and saving** > Click on **Data saving** and choose *None (test mode)*. The window should look like figure 4.



Specify storage during the DAQ job and saving afterwards.

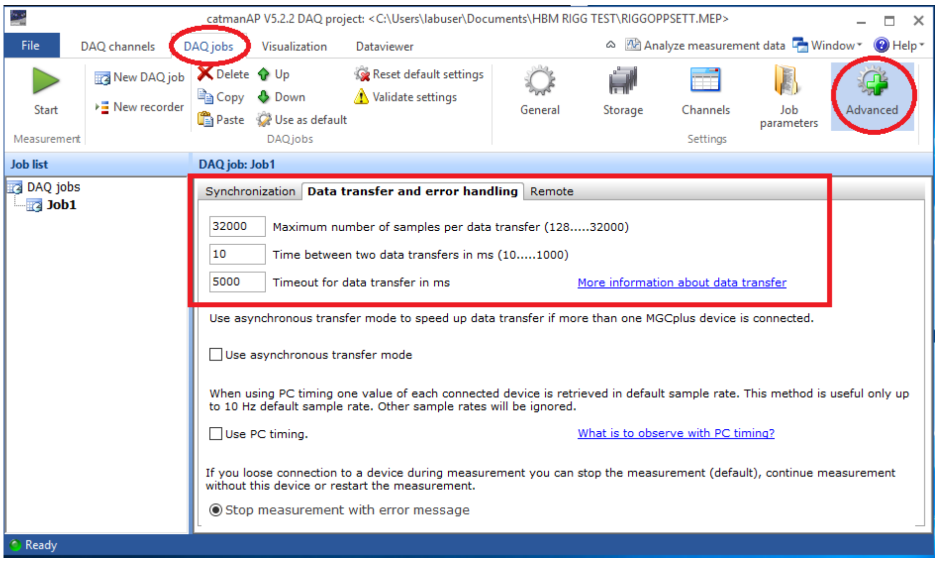
Storage mode: Keep all data
Data saving: None (test mode)
File format: MS Excel Office 2007 XML
Precision: 8-Byte
Saving depth: Store all data

Saving file
Placeholder | C:\HBM TESTING\test1.XLSX

Figure 4: Storage management in Catman

5.2.4 Transfer

The size and frequency of data transmissions can be managed. To do this you need to:
Go to **DAQJobs** in the header > Choose **Advanced** and then **Data transfer and error handling**.
The window should look like figure 5.



32000 Maximum number of samples per data transfer (128.....32000)

10 Time between two data transfers in ms (10.....1000)

5000 Timeout for data transfer in ms — More information about data transfer

Use asynchronous transfer mode to speed up data transfer if more than one MGCplus device is connected.

☐ Use asynchronous transfer mode

When using PC timing one value of each connected device is retrieved in default sample rate. This method is useful only up to 10 Hz default sample rate. Other sample rates will be ignored.

☐ Use PC timing. — What is to observe with PC timing?

If you loose connection to a device during measurement you can stop the measurement (default), continue measurement without this device or restart the measurement.

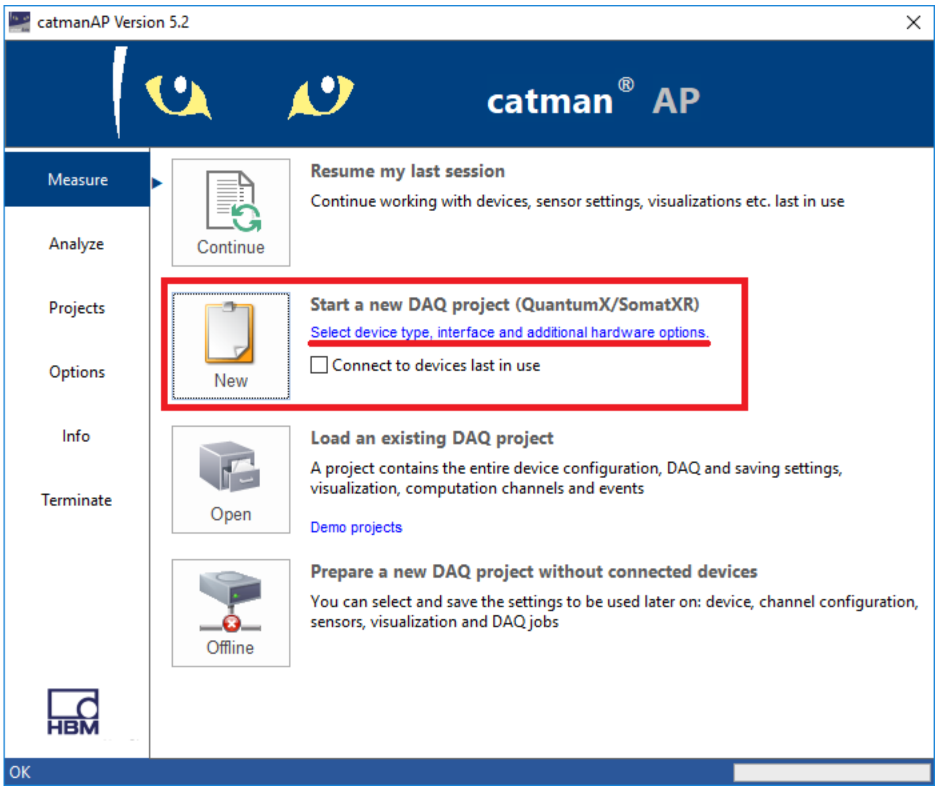
◉ Stop measurement with error message

Figure 5: Transfer management in Catman

5.2.5 Create New Project (OPTIONAL)

To create a new project file (`.MEP`): open Catman AP (See figure 6) > Click on "**Select device type, interface and additional hardware options**" > In this new window (See figure 7) Click on **Hardware time channels**, choose **Create hardware time channels** and click **OK** > Click on **Start a new DAQ project** > In this new window (See figure 8) click **Connect**.

Note that the data acquisition board must be connected to the sensors and the computer with Catman for this to work properly.



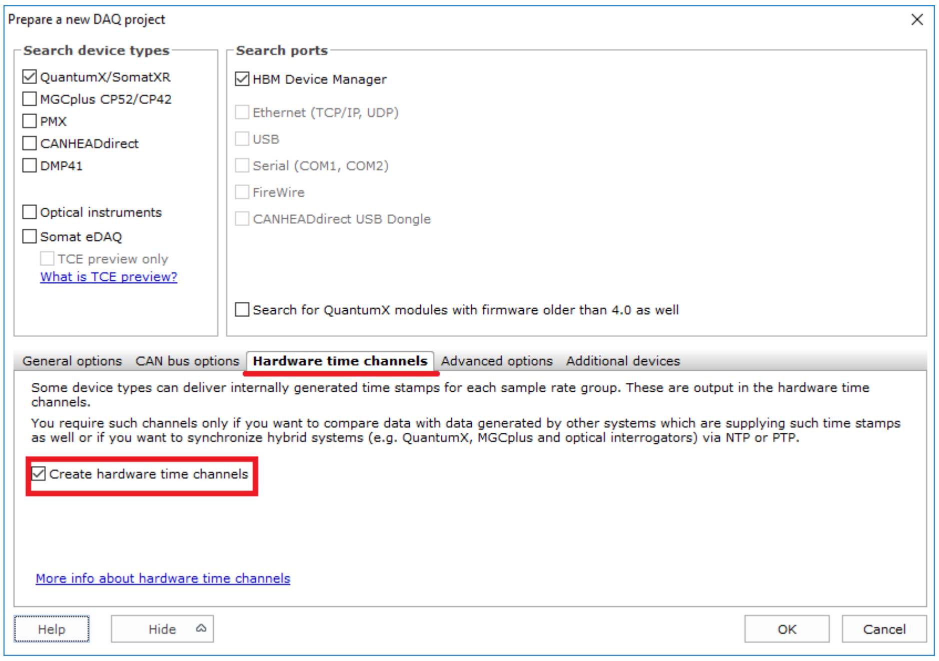
Resume my last session
Continue working with devices, sensor settings, visualizations etc. last in use

Start a new DAQ project (QuantumX/SomatXR)
Select device type, interface and additional hardware options.
☐ Connect to devices last in use

Load an existing DAQ project
A project contains the entire device configuration, DAQ and saving settings, visualization, computation channels and events
Demo projects

Prepare a new DAQ project without connected devices
You can select and save the settings to be used later on: device, channel configuration, sensors, visualization and DAQ jobs

Figure 6: New Project in Catman AP (1)


Prepare a new DAQ project

Search device types

- [✓] QuantumX/SomatXR
- [ ] MGCplus CP52/CP42
- [ ] PMX
- [ ] CANHEADdirect
- [ ] DMP41
- [ ] Optical instruments
- [ ] Somat eDAQ
  - [ ] TCE preview only

What is TCE preview?

Search ports

- [✓] HBM Device Manager
- [ ] Ethernet (TCP/IP, UDP)
- [ ] USB
- [ ] Serial (COM1, COM2)
- [ ] FireWire
- [ ] CANHEADdirect USB Dongle
- [ ] Search for QuantumX modules with firmware older than 4.0 as well

General options | CAN bus options | **Hardware time channels** | Advanced options | Additional devices

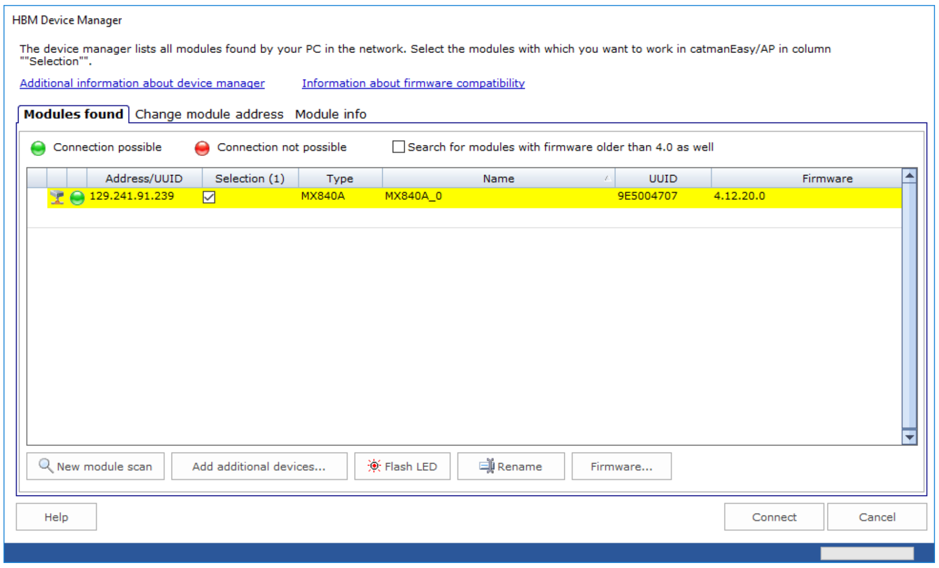
Some device types can deliver internally generated time stamps for each sample rate group. These are output in the hardware time channels.

You require such channels only if you want to compare data with data generated by other systems which are supplying such time stamps as well or if you want to synchronize hybrid systems (e.g. QuantumX, MGCplus and optical interrogators) via NTP or PTP.

- [✓] Create hardware time channels

More info about hardware time channels

Figure 7: New Project in Catman AP (2)


HBM Device Manager

The device manager lists all modules found by your PC in the network. Select the modules with which you want to work in catmanEasy/AP in column ""Selection"".

Additional information about device manager   Information about firmware compatibility

Modules found | Change module address | Module info

🟢 Connection possible   🔴 Connection not possible   ☐ Search for modules with firmware older than 4.0 as well

Address/UUID	Selection (1)	Type	Name	UUID	Firmware
129.241.91.239	✓	MX840A	MX840A_0	9E5004707	4.12.20.0

Figure 8: New Project in Catman AP (3)

5.3 Server

This section explains how to set up the cloud software on the server from scratch.
NOTE:

- Before you start you need to install Python and Node with NPM on the server.
- The servers Firewall may have to be configured to allow for UDP communication on ports 8001 and 7331, and TCP communication on port 1337.
- To gain access to the folder *DT_Example* you must sign a non-disclosure agreement (NDA) with SAP.
- The *udpplotter* can be retrieved from the Github repository: "https://github.com/simennj/udpplotter". It is currently private because of license restrictions.
- You need to start a job in Catman for the plotting to commence. To do so simply press the `Start`-button found in the top-left corner in the Catman window.

- Catman has to be set up to send the data to the new server, see 5.2.2.

Procedure:

1. Install all necessary Python packages. A complete list of the packages can be found in Appendix B.
2. Navigate to the directory of the *udpplotter* folder (see notes) in the terminal and type `npm install`.
3. Run the web application by typing `node index.js` in the terminal.
4. In a new terminal window navigate to the *DT_Example* folder (see notes) and run the command: `python RigSolver.py`

The server should now be set up properly. If you have configured the rest of the system according to sections 5.1 and 5.2 you should now be able open the web application if you type `localhost:1337` in the server's web browser. If the firewall is set up correctly, the web application should then be available on `<server address>:1337` on other computers.

6 Discussion and Evaluation

6.1 Technologies

6.1.1 Data Acquisition System

A data acquisition system consists of three parts: sensors, data acquisition boards and data acquisition software. The sensors capture and quantify a physical phenomena through a voltage which is then sampled by a data acquisition board. The samples are read by the data acquisition software and the voltage value is translated into a corresponding engineering unit. Examples of DAQ software is Catman by HBM and LabVIEW by National Instruments.

At the beginning of the project, a previous setup was available using a data acquisition board from HBM and Catman, and there was no immediate need for changes. However, in the early stages of the project the license for Catman expired. An alternative to purchasing license based data acquisition software is to develop an in-house software solution. In addition to cost savings, an in-house software solution is more transparent and can offer more control than Catman. The possibility of an in-house software solution was explored and a prototype was developed. This prototype was able to retrieve raw data from the data acquisition board.

While the prototype is able to retrieve the raw data from the board, there are still two obstacles. The first is interpreting data from the data acquisition board; what values are received and which sensor they originate from. The second is mapping the voltage values to an engineering unit. Both of these issues require access to documentation of the sensors and the data acquisition board in order to be solved.

At that point there were two clear ways forward, either continue working on the prototype or renewing the Catman license. After discussing the options with Terje Rølvåg it was decided to renew the Catman licence. This was due to time constraints and uncertainty of successfully finishing the prototype without access to the proper documentation. However, we would like to stress that the digital twin cloud software is not locked to Catman.

The choice of data acquisition solution should be assessed in the case of instrumenting a new physical asset. As long as the solution is able to send the measured values as doubles through UDP, it should be compatible with the digital twin solution. This could potentially reduce costs spent on hardware and software licenses.

6.1.2 Server Architecture

One of the sub-goals of the project is to be able to host digital twin software externally in an application. During development, two options have been considered: Self hosting and renting space at cloud computing service companies such as Amazon Web Services (AWS), Microsoft Azure or NTNU IT. Hosting at a cloud service required less work than self hosting and was therefore preferable. After researching the cloud services we discovered that while AWS and Azure are expensive, hosting at NTNU IT would not cost anything and still provide the necessary features. The chosen solution was to host a local VM provided by NTNU IT.

6.1.3 Data Communication

Two protocols for sending raw data over the internet were considered: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). An assessment was done in order to choose which would be the best for the digital twin cloud software. For this project, the most important difference between the protocols is that UDP simply sends the packets without checking if they are received while TCP re-sends the packet if it is not received. It was decided after discussions with Terje Rølvåg that in the case of a lost packet it would be better to continue transmitting new packets instead of halting the stream to re-transmit the lost packet. The need for a high throughput with as little delay as possible is deemed more important than the occasional loss of a packet. Since the latter has no noticeable effect on the simulation, UDP was chosen as the data communication protocol.

6.1.4 Visualisation tools

There is a large number of visualisation tools for web development available that offer 2D graphics, both open-source and closed-source. However, for the digital twin cloud software we needed a tool that could make a 2D-plot of a live data-stream, without stuttering. To avoid losing time on issues regarding licenses, open-source libraries were prioritised. After research and testing, the JavaScript library `Plotly` was chosen. Other tools were reviewed, but due to the successful implementation of `Plotly` we chose not to go any further with other options.

The number of visualisation tools for web development that offer 3D graphics is more limited. There are a few open-source libraries such as `BabylonJS` and `Three.js` specifically made for 3D graphics, but they do not support FE-models. We were introduced to the company Ceetron by Terje Rølvåg which offers several tools for visualisation and post-processing of FE-models. A meeting was arranged with Ceetron and SAP in late November to discuss how we could use Ceetron software in our web application to visualise and animate the FE-model results. For this purpose it was suggested that we make use of the `Unstruct Surface Grid` (USG) model functionality found in `Ceetron Cloud Components`.

Ceetron also suggested an alternative solution. It required an additional server component, and was more complex. USG was therefore chosen since the additional functionality from the other solution was not required for this project. Swapping to the more complete solution was described as being a feasible option, if functionality not offered in USG is required in the future.

6.2 Challenges and limitations

In the beginning of this project we were introduced to three different physical assets: The Torsion Bar Suspension Rig, a crane located at MTP laboratories at Valgrinda and Lerkendal stadium. All three physical assets lacked the necessary hardware components for this project. The computer located on the Torsion Bar Suspension Rig had recently broken down, but a new one had been ordered. The crane at Valgrinda lacked a data acquisition board and a computer, while Lerkendal stadium lacked all the hardware components. In order to start prototyping as soon as possible we decided to start working with the Torsion Bar Suspension Rig since it required the least time to get up and running. In addition it was the asset that was most accessible and complete.

Digital Twin as a field and as a concept is still in the process of being established and developed. As a result, there are very few 'best practices' available. In discovering what tools to employ there was thus very little documentation regarding how to utilise them. This extended to FEDEM and Catman, where the complexity of the programs and lack of proper documentation of the relevant functionality have been a challenge. An example of this was during our first attempt at streaming the incoming data through FEDEM. The Dynamic Link Library (DLL) for the FEDEM solver exposed only the name of the functions with no explanation of their input parameters, types or purposes. Since there was no documentation or header files available we were unable to use it directly and had to use a wrapper from SAP, which was not immediately available. Catman had similar issues with documentation, especially regarding the physical wiring needed for the remote connection option. It was eventually solved by trial and error.

Another challenge was selecting which tools to employ and when. While there is no established best practice in cloud software for digital twins, there are plenty of tools that advertise as being helpful. There are many streaming analytics tools which claim to 'process continuous streams of event data in real time and act on the results'. During development, some of these tools were tested (SAP Analytics Cloud for instance). However it was decided that for now we would not utilise these tools as most of the analysis needed could be handled by simple statistics and plots.

6.3 Scalability

The server currently runs on a virtual machine with limited resources. This puts a limit on how many processes and script jobs that can run simultaneously. Consequently, in order to support a larger user base than the MTP department, one would need more space and processing power, especially if more complex analytic tools are needed later on. These tools will likely require the ability and space to store historical data, as currently data may only be stored client side.

Additional resources could be granted from NTNU IT if necessary. Moving the solution to a different host with more resources is also possible.

6.3.1 Adding a new digital twin

Our digital twin cloud software is tailored towards the Torsion Bar Suspension Rig and there is currently no functionality to simply add new models. Most of the code on the server can be reused (Listing 1, 2, 3 and 4) for a new model. However, there are lines of code that are model specific and these will mainly depend on:

- Number and types of sensors
- Output format for sensor data (See table 1)
- Configuration of external functions in FEDEM model
- Which values should be plotted

Should the new physical asset in question be equipped with another data acquisition system than described in section 6.1.1 this should not present a problem. As long as the data acquisition system uses UDP to send sensor values as a byte stream, the system will work with only minor adjustments on the server side.

6.4 Further work

As mentioned in section 6.2 there were two additional assets that could be used. An advisable task would be to instrument at least one of these assets. This will be beneficial for two reasons: First, if the instrumentation process is documented well, the documentation can be used as guide for setting up data acquisition systems for other physical assets later. Second, it will make it possible to test the robustness and scalability of the current digital twin cloud software.

A live video stream of the physical twin in the client is a requested feature. This feature would make it easier to verify that the digital twin behaves the same way as the physical. The system currently requires a computer at the site of the physical twin. Therefore, a solution is to connect a camera to the computer and send the live stream to the server in a similar fashion as the sensor data.

Another requested feature is event triggers to reduce the amount of uninteresting data received. In digital twins, only some of the behaviour will be of relevance, i.e during activity and under stress.

Currently the web application is tailored to visualise the Torsion Bar Suspension Rig. In the future, a more flexible visualisation setup is desired to make transition between different digital twins simpler for both the user and the developer. The visualisation should also be expanded to show deformation and stress in the form of colour change in the 3D model. The stress could in addition be visualised by a S-N curve as part of Fatigue analysis, however that would likely be separate from the current visualisation.

For digital twins equipped with accelerometers, a key feature to implement would be Fast Fourier Transform. This enables frequency analysis of the asset and can be used to detect structural changes. In addition it can be used to verify the precision of the FE model.

7 Conclusion

Cloud-based solutions for digital twin modelling have been explored and an environment has been established for developing a software solution. Requirements have been specified for developing a cloud based digital twin software solution. A prototype based on the torsion bar suspension rig has been created showcasing and satisfying most of the major points of the requirements. A user guide for how to setup each component of the prototype is available for reproducing or referencing the current system. Steps have been outlined for further iteration on this prototype to move towards a complete digital twin cloud solution.

Appendices

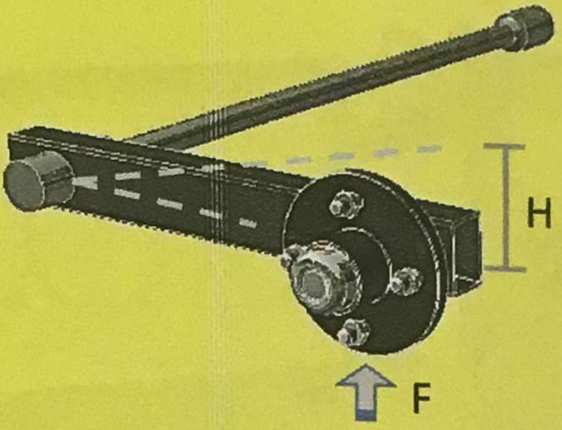
A Torsion Bar Suspension Rig Manual

Physical Test Manual

Torsion Bar Suspension Rig

Maximum Capacity

The rig is fail-proof. This means that it is OK to elevate the hydraulic jack to maximum stroke-length.

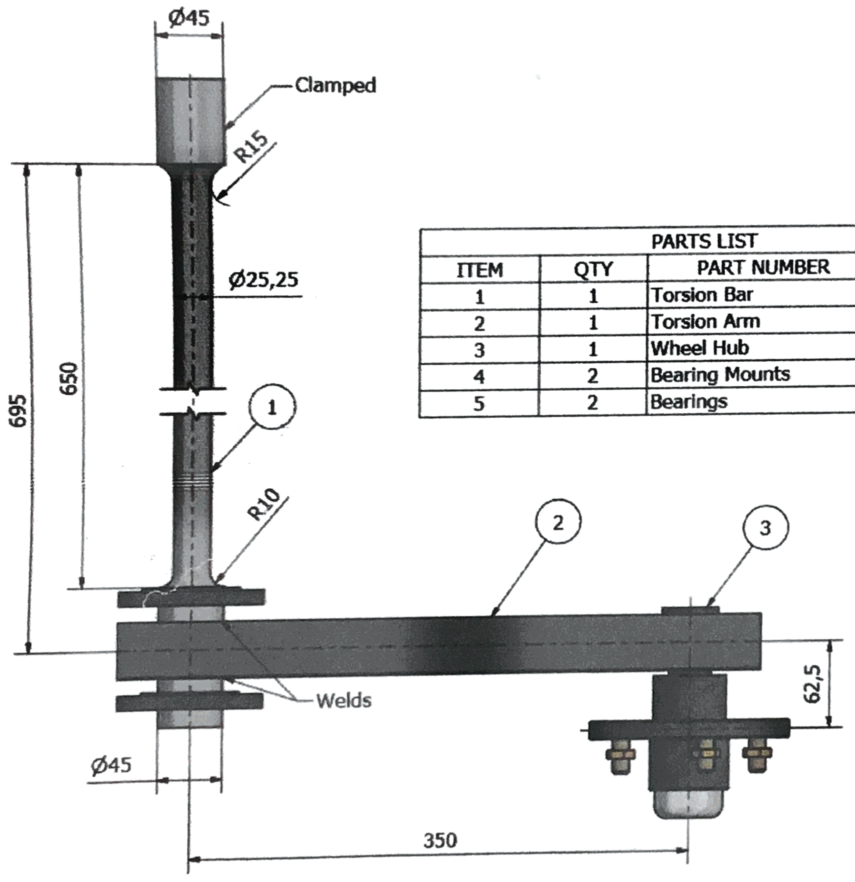


Deflection Angle	α	12,5	degrees
Elevation Height	H	81	mm
Force	F	3300	Newtons

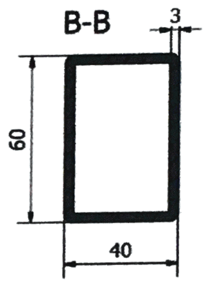
Torsion Bar Material Data

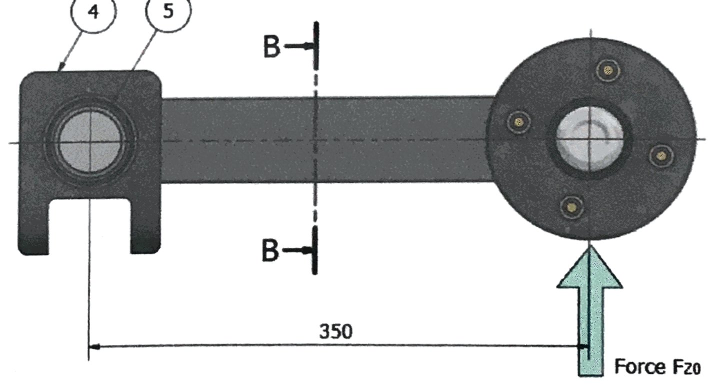
Stainless Steel - SS2387 / S165M

E-modulus	E	210	Gpa
G-modulus	G	78	Gpa
Yield Strength	σ_{ys}	885	Mpa
Ultimate Tensile Strength	σ_{ut}	1010	Mpa
Density	δ	7700	Kg/m
Fatigue Strength Coefficient	σ'_f	1454	Mpa
Fatigue Strength Exponent	b	-0,08	
Fatigue Ductility Coefficient	ϵ'_f	1.85	
Fatigue Ductility Exponent	c	-0,72	
Cyclic Yield Strength	σ'_s	716	Mpa
Cyclic Strength Coefficient	K'	1367	Mpa
Cyclic Strain Hardening Exponent	n'	0,10	
Notch Sesitivity Factor	K_f	1,14	



PARTS LIST			
ITEM	QTY	PART NUMBER	MASS
1	1	Torsion Bar	4,7 kg
2	1	Torsion Arm	1,8 kg
3	1	Wheel Hub	3,7 kg
4	2	Bearing Mounts	0,6 kg
5	2	Bearings	0,2 kg

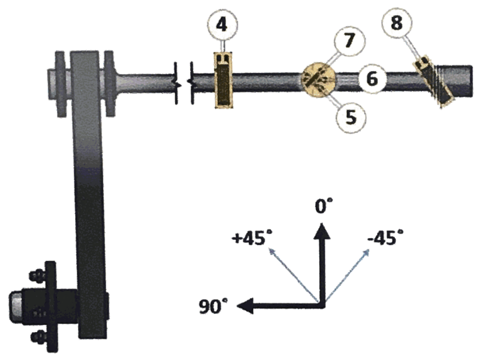


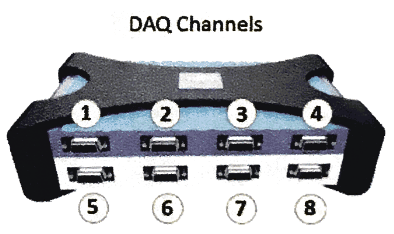


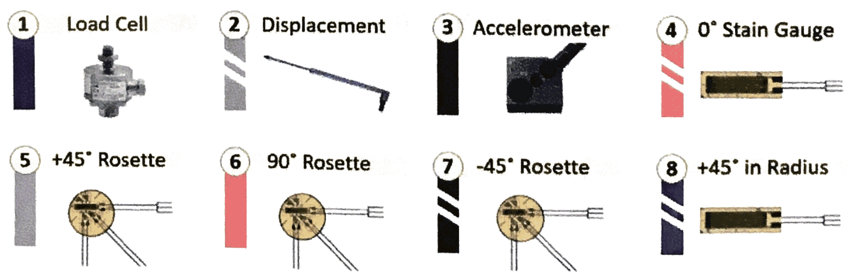
Sensors & Equipment

The rig is equipped to examine quasi-static response and the Eigen frequency.

8 sensors are installed on the rig. They are connected to a Data Acquisition box (DAQ) connected to the computer to conduct live monitoring of the tests. The applied Force is monitored by a Load Cell situated directly above the hydraulic jack. A displacement probe monitors the elevation of the wheel hub. An accelerometer is used to detect the dynamic response (Eigen Frequency) of the suspension system. Five strain gauges are situated in different locations and angles on the torsion bar. These are used to compute the torsion bar stresses.

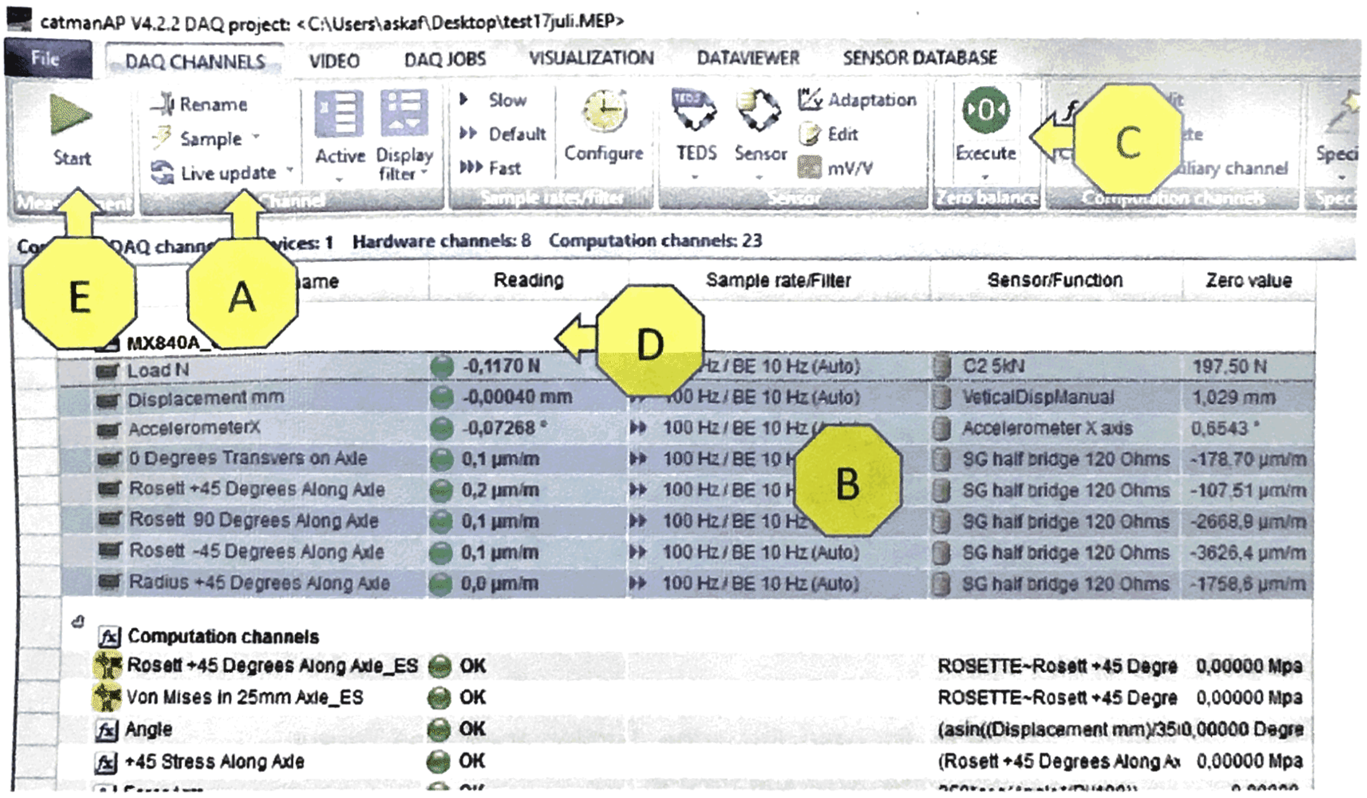






Quasi-static Test Manual

1. **Open Catman AP**
2. **Click: *Continue* (Resume my last session)**
   This opens the DAQ Channels window. Here, all the active sensors are displayed.
3. **Lower the hydraulic so the wheel hub moves freely.**
4. Before initiating a test, the sensors needs to be calibrated and zeroed. Due to the weight of the torsion arm and wheel hub (46N), this needs to be accounted for.
   Click on "A" *Live Update*. This enables live readings of the sensors. The values are visible in the Reading-column.
   Mark all the 8 sensors, "B".
   Click "C" *Execute*, to zero all the values.
   Elevate the hydraulic jack slowly, until the Load reads 46N, "D".
   Click "C" *Execute*, to zero all the values again.
5. **Click "E" *Start*** to initiate the test.

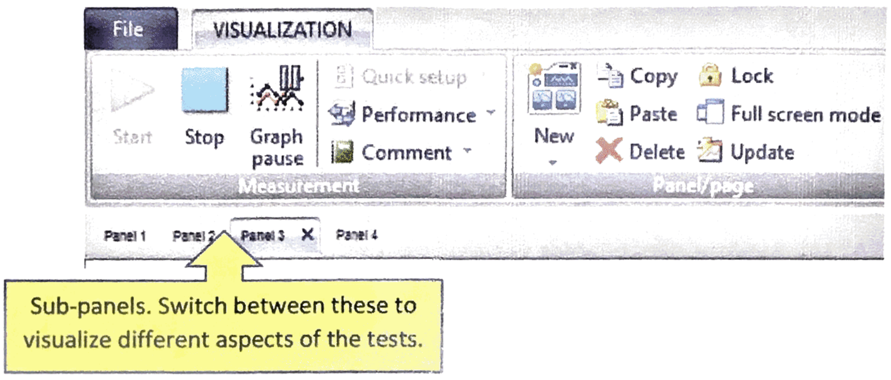


catmanAP V4.2.2 DAQ project: <C:\Users\askaf\Desktop\test17juli.MEP>

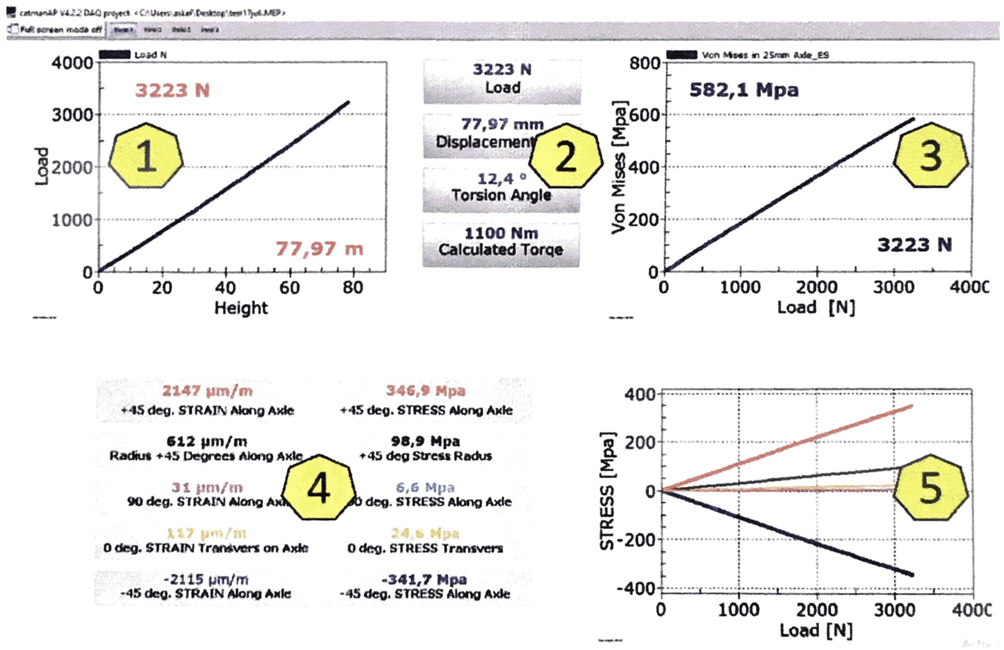
DAQ chann… vices: 1 Hardware channels: 8 Computation channels: 23

ame	Reading	Sample rate/Filter	Sensor/Function	Zero value
MX840A_				
Load N	-0,1170 N	Hz / BE 10 Hz (Auto)	C2 5kN	197,50 N
Displacement mm	-0,00040 mm	00 Hz / BE 10 Hz (Auto)	VeticalDispManual	1,029 mm
AccelerometerX	-0,07268 °	100 Hz / BE 10 Hz	Accelerometer X axis	0,6543 °
0 Degrees Transvers on Axle	0,1 µm/m	100 Hz / BE 10 H	SG half bridge 120 Ohms	-178,70 µm/m
Rosett +45 Degrees Along Axle	0,2 µm/m	100 Hz / BE 10 H	SG half bridge 120 Ohms	-107,51 µm/m
Rosett 90 Degrees Along Axle	0,1 µm/m	100 Hz / BE 10 Hz	SG half bridge 120 Ohms	-2668,9 µm/m
Rosett -45 Degrees Along Axle	0,1 µm/m	100 Hz / BE 10 Hz (Auto)	SG half bridge 120 Ohms	-3626,4 µm/m
Radius +45 Degrees Along Axle	0,0 µm/m	100 Hz / BE 10 Hz (Auto)	SG half bridge 120 Ohms	-1758,6 µm/m
Computation channels				
Rosett +45 Degrees Along Axle_ES	OK		ROSETTE~Rosett +45 Degre	0,00000 Mpa
Von Mises in 25mm Axle_ES	OK		ROSETTE~Rosett +45 Degre	0,00000 Mpa
Angle	OK		(asin((Displacement mm)/35	0,00000 Degre
+45 Stress Along Axle	OK		(Rosett +45 Degrees Along Ax	0,00000 Mpa

6. When a test has been started, the *VISUALIZATION panel opens.* This panel gives live monitoring of the test. **Sub-panels are prepared to visualize the quasi-static testing. Switch between these to visualize different aspects of the tests.**



Sub-panel 1 – Stress and Overview



2147 µm/m
+45 deg. STRAIN Along Axle

346,9 Mpa
+45 deg. STRESS Along Axle

612 µm/m
Radius +45 Degrees Along Axle

98,9 Mpa
+45 deg Stress Radus

31 µm/m
90 deg. STRAIN Along Axle

6,6 Mpa
90 deg. STRESS Along Axle

117 µm/m
0 deg. STRAIN Transvers on Axle

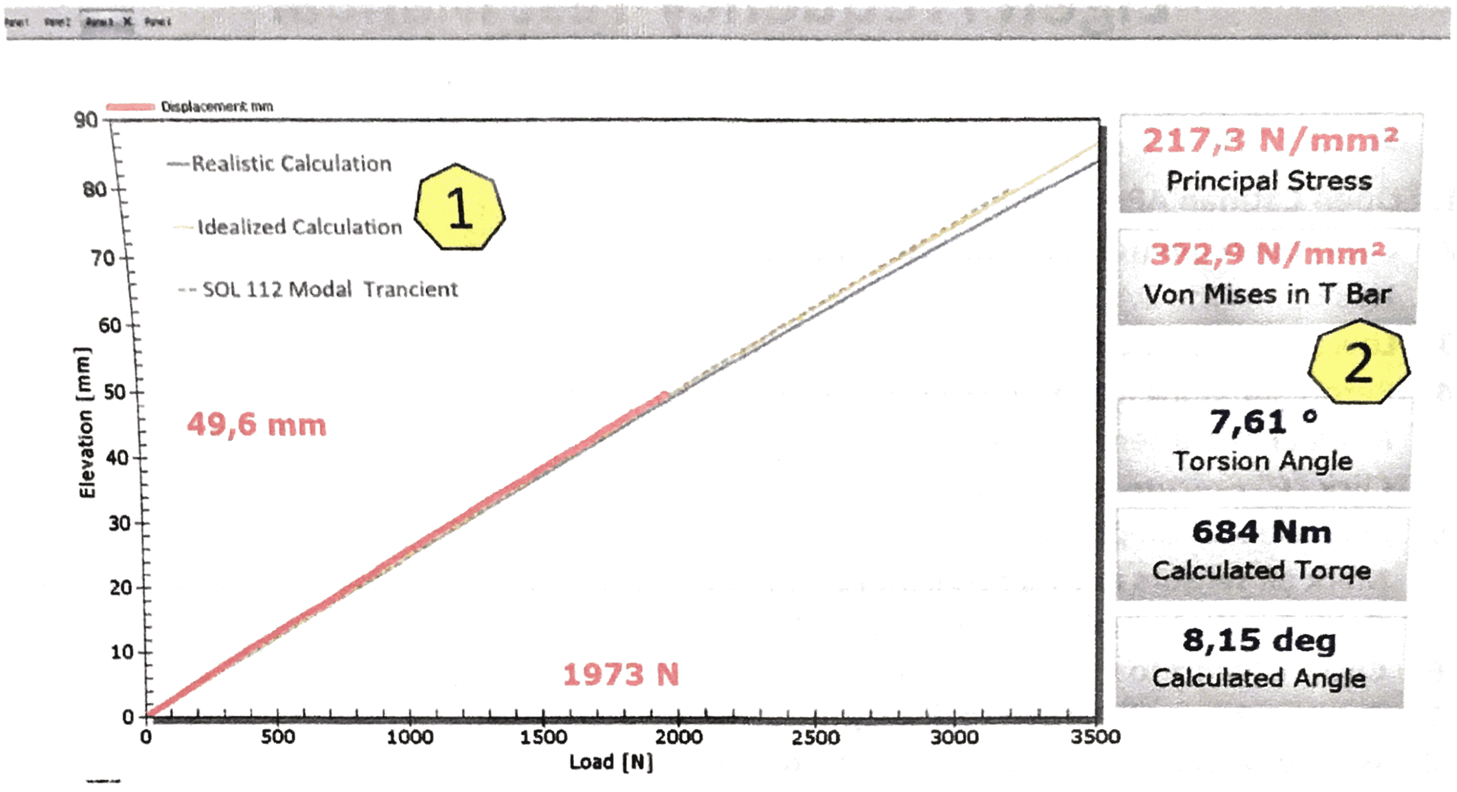
24,6 Mpa
0 deg. STRESS Transvers

-2115 µm/m
-45 deg. STRAIN Along Axle

-341,7 Mpa
-45 deg. STRESS Along Axle

(1) Elevation height/Load.
(2) Values for Load, Elevation, Torsion angle, and Torque.
(3) Von Mises Stress in torsion bar.
(4) Strain in every strain gauge and corresponding stress.
(5) Visualization of the stresses.

Sub-panel 2 Elevation vs applied load



(1) Elevation height/Load. The Background picture displays analytical and virtual solutions to estimate the height/load relationship.

(2) Values: Principal and Von Mises stresses in the torsion bar. Torsion angle, Torque and calculated torsion arm angle.

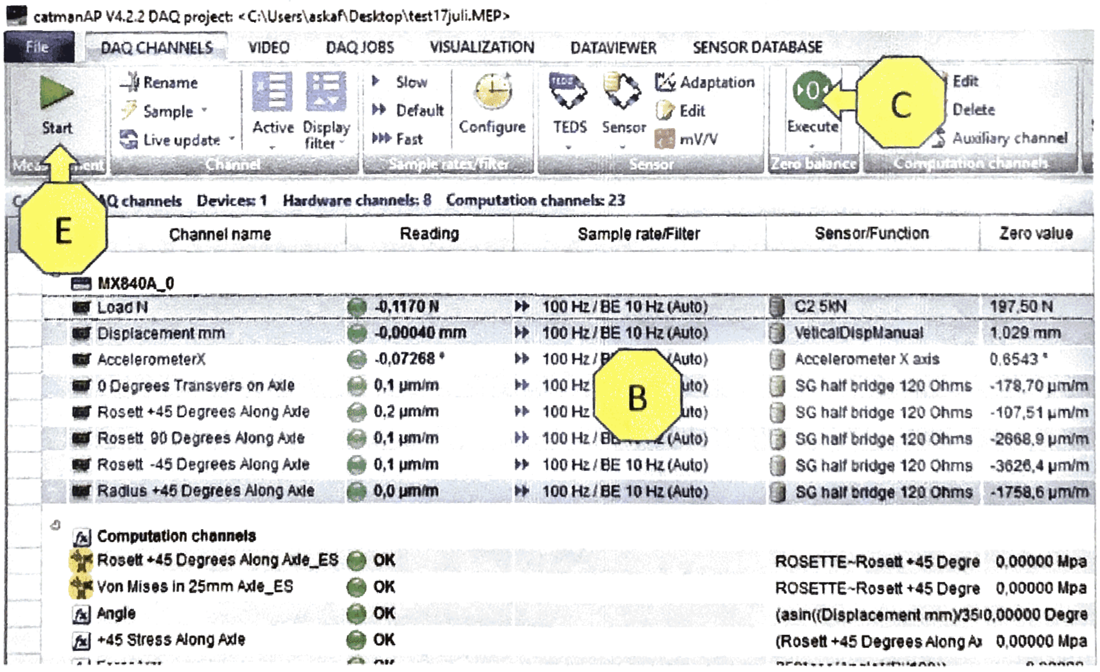
7. **Use the hydraulic jack to elevate the wheel hub.** Watch the Live monitoring.

8. **When hydraulic jack reaches the maximum position, Click *Stop* to end the test.**

9. **Test data can be exported by selecting:** *File →Save as→Save last DAQ job.* Choose desired format (e.g. Excel or Matlab)

Eigen Frequency Test Manual

1. **Open Catman AP**
2. **Click: *Continue* (Resume my last session)**
   This opens the DAQ Channels window. Here, all the active sensors are displayed.
3. **Lower the hydraulic so the wheel hub moves freely.**
4. Before initiating a test, the sensors needs to be calibrated and zeroed.
   Mark all the 8 sensors, "B".
   Click "C" *Execute*, to zero all the values.
5. Increase the sample rate:
   Mark all sensors and Right-click directly above the sample rate,"B". Click *Configure Sample Rate.* Set the sample rate to at least 300Hz.
6. **Click "E" *Start*** to initiate the test.
7. **Select sub-panel; Panel 4 to display the dynamic visualization.**
8. **Hit the wheel hub by hand repeatedly to initiate oscillation.** A sprike on the right graph will occour. This identifies the Eigen frequency.
9. **If desired: Attach the extra wheel hub weight to examine the difference.**
10. **End the test by clicking *Stop.***
11. **Reset the sample rate to 100Hz.**



catmanAP V4.2.2 DAQ project: <C:\Users\askaf\Desktop\test17juli.MEP>

AQ channels Devices: 1 Hardware channels: 8 Computation channels: 23

Channel name	Reading	Sample rate/Filter	Sensor/Function	Zero value
MX840A_0				
Load N	-0,1170 N	100 Hz / BE 10 Hz (Auto)	C2 5kN	197,50 N
Displacement mm	-0,00040 mm	100 Hz / BE 10 Hz (Auto)	VeticalDispManual	1,029 mm
AccelerometerX	-0,07268 °	100 Hz / B (Auto)	Accelerometer X axis	0,6543 °
0 Degrees Transvers on Axle	0,1 µm/m	100 Hz uto)	SG half bridge 120 Ohms	-178,70 µm/m
Rosett +45 Degrees Along Axle	0,2 µm/m	100 Hz uto)	SG half bridge 120 Ohms	-107,51 µm/m
Rosett 90 Degrees Along Axle	0,1 µm/m	100 Hz / B (Auto)	SG half bridge 120 Ohms	-2668,9 µm/m
Rosett -45 Degrees Along Axle	0,1 µm/m	100 Hz / BE 10 Hz (Auto)	SG half bridge 120 Ohms	-3626,4 µm/m
Radius +45 Degrees Along Axle	0,0 µm/m	100 Hz / BE 10 Hz (Auto)	SG half bridge 120 Ohms	-1758,6 µm/m
Computation channels				
Rosett +45 Degrees Along Axle_ES	OK		ROSETTE~Rosett +45 Degre	0,00000 Mpa
Von Mises in 25mm Axle_ES	OK		ROSETTE~Rosett +45 Degre	0,00000 Mpa
Angle	OK		(asin((Displacement mm)/35	0,00000 Degre
+45 Stress Along Axle	OK		(Rosett +45 Degrees Along Ax	0,00000 Mpa

B Software Packages

B.1 Node Packages

- python-struct
- dgram
- express
- http
- socket.io

B.2 Python Modules

- struct
- socket
- vpmSolverRun
- vpmSolver

Simen Norderud Jensen

Building an extensible prototype for a cloud based digital twin platform

NTNU
Norwegian University of
Science and Technology