DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Visual Configuration and Debugging of Distributed Applications

Thomas Ossowski

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Visual Configuration and Debugging of Distributed Applications

Visuelle Konfiguration und Debugging von Verteilten Anwendungen

Author:	Thomas Ossowski
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	M. Sc. Sandro Weber
Submission Date:	15.10.2021

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.10.2021 Thomas Ossowski

Acknowledgments

Now that this work has been completed, I don't want to miss thanking a few people. First of all, I would like to thank Prof. Gudrun Klinker, Ph.D., who made it possible to write this thesis at her chair. I also want to thank M.Sc. Sandro Weber. He explained UBII to me and gave me implementation and design hints that made this work possible. He always took the time when needed and answered all questions asked. Last but not least, I would like to thank my family and friends who gave me the leeway to finish this work within the time frame.

Abstract

A typical model for distributed applications in Software-Architecture is a client-server model. This model proposes to partition tasks between a server and clients. Clients are able to request resources from the server or consume a stream of data coming from a service running on the server over the network. This architecture is now being challenged by the cloud or edge infrastructure where tasks or applications are distributed over the network, not having just one server but multiple producers. Such systems are more challenging to maintain just because of the fact that there is not only one endpoint like the server in the client-server model but multiple endpoints. This work will propose an editor capable of visualizing such a system with the help of graphs and also the ability to debug distributed applications.

Contents

1 Introduction

This chapter first introduces the rationale behind this thesis, followed by a rough description of the problem, as well as a short overview of the subsequent chapters.

1.1 Motivation

Whenever a device is purchased, the inevitable question arises of how to communicate with the device to enable control. For example, if a computer is purchased, at least the acquisition of a keyboard or a mouse must be considered. These two devices are already sufficient to enable a variety of interaction possibilities with the computer. As versatile as the mouse is as an input device, it too has its limits when either more complex inputs have to be made or the location makes input with the mouse difficult or not possible. It is clearly disadvantageous to control your mobile phone with a mouse. Furthermore, there are plenty of additional input devices that allow new interaction possibilities, which are not possible to achieve with only mouse or keyboard. An example would be a camera that capture and recognise certain predefined features of the body with the help of image recognition methods. Like for example the direction of the eyes.

In the above scenario, there are several problems that the device must try to handle. While there are efficient methods for computers to process actions of the mouse or keyboard, the recognition of the direction of the eyes with only the help of a camera requires a complex procedure which effectively requires computational effort. Another feature that would be desirable is the reuse of a peripheral device for the input or an already existing algorithm e.g. for enabling voice comprehension for multiple devices in a network. A typical example is the server that can be operated via a SSH connection with the keyboard connected to a Client. For this and more the Ubi-Interact Framework (UBII) was developed.

UBII is a network of clients which have devices or processing modules. Devices can send signals or topics to other clients hosting processes which then process them and produce a desired output which then can be send to another Client. Broadly speaking, UBII is a Manager of distributed application hosted on Clients having Devices and Processing modules. Such applications can quickly become relatively complex and require an intuitive form of how to configure and program the Clients, Devices and Processing modules in a network. This thesis propose a solution on how UBII can be visualized and configured by an Editor utilizing a Graph. Then, smaller components are presented to facilitate debugging the applications hosted on UBII.

1.2 Main Contributions

The Main Contribution of this thesis was developing a web application which visualize an UBII Session and give the possibility to debug the application. The corresponding research question is: How can a network of distributed applications be visualized intuitively and what can be done in order to debug such applications.

During this Thesis a graph editor was designed and build with the help of several JavaScript (js) Libraries. The work was evaluated permanently by an expert and therefore refined constantly. Also multiple features were added, so it is possible to explore an application hosted on UBII in order to find errors. Then the work was integrated into the UBII-Web Framework. After the realization the Editor was tested and evaluated on a real and on a pseudo Application.

1.3 Outline

This chapter describes the concept of this thesis.

First, in **Chapter 2** some basics of distributed applications, Human Computer Interactions (HCI), networking as well as the tools used in this work are briefly discussed, furthermore the concept of UBII is explained, for which this work was designed. In **Chapter 3** research results in the area of graph theory and debugging are discussed.

In **chapter 4** the realization of the graph editor is discussed in detail, especially decision processes, design and also derivatives that didn't qualified for further investigation. This is followed by an performance analysis of the editors capabilities in terms of rendering speed. At the end the application was tested with an application running on UBII.

In **Chapter 5** the implemented features which should be helpful for error detection are presented. Two features in particular are highlighted. One is about analyzing messages that applications hosted on UBII sends for further processing. The second feature is about the reachability of the clients. This is followed by an performance analysis of how many inputs or outputs an Processing Module can have in order to still run the editor.

In **Chapter 6** a conclusion is drawn, as well as a discussion about possible extensions or improvements.

2 Background

This chapter describes the given scenario. It first discusses distributed systems and the connection between distributed applications and UBII. In order to understand the topics, a short introduction to Human-Computer-Interaction (HCI) is given. This is followed by a short introduction to graphs. It follows an explanation of how errors in a distributed system can be found in order to help users finding bugs. Finally, technologies for the browser that came into question for this work are shortly discussed.

2.1 Distributed Applications and Ubiquitous Interact (UBII)

2.1.1 Distributed Applications

A proven pattern in software engineering is the server-client model. In this model, a server provides resources or data that can be requested by clients. For most scenarios this pattern is suitable for a lot of applications, but this model has several disadvantages. On the one hand it can happen that the server is overloaded and therefore no longer functional, because it has received too many request from too many clients. Furthermore, there is only one way to run processes in concurrency in this architecture, e.g. by running threads from the CPU. An example where an application hosted on one server is already on the limit of functioning is image recognition, especially when multiple clients are requiring data in a concurrent manner.

In contrast to this model an application can be hosted as a distributed systems. Here software is distributed in the network and applications communicate with each other over the network in order to share data. This expose several problems and/or advantages. On the one hand the modules of an application run in competition with each other. The advantage is that for some problems a speedup can be achieved, but this also requires a coordination on how data from one module of the application can be shared with other modules. There is also another handicap. On a single CPU the application can rely on a global time, on the other hand and application hosted on distributed system can not utilize a global time, which makes sharing data even more difficult. Furthermore specialized systems can be created, whose hardware is aligned to certain problems which can be categorized. Sadly a distributed application is prone to a group of new errors yielding from distributed applications. For example a module running on a system can fail which is needed for a process on another system, or the connection to the client can abort or the connection is slower than the capability of processing data of another module running on a different client. Finding such errors is difficult and require most of the time additional tools which are designed for finding such errors. More can be found here [Cou+12].

2.1.2 UBII

UBII is a distributed system. Figure 2.1 In UBII, clients can communicate with each other in a network. Clients can have processing modules or devices. A device is e.g. a computer mouse or a keyboard, in order to use a device it needs to be connected to a Client. Clients in UBII are used to log on to the network. If a client becomes inactive at some point, i.e. if a client no longer responds to requests from UBII, the CLIENT logs off from the UBII network with all its devices. Devices can have components. Components are the gate for inputs and outputs. In the mouse example, the mouse can have an output for distributing the sensor information. In contradiction the inputs can listen to data. Processing Modules are relatively similar to ordinary functions wich can be written in multiple languages like C++ or JavaScript, but can be run as a service on a Client. Processing Modules can have a clock frequency, i.e. a how fast data should be processed. A Processing Module can have both inputs and outputs, in UBII inputs can subscribe to Topics and outputs can publish Topics to interested clients. The information, which clients, devices, processing modules offer or demands can be set on a session.Since all participants are in a network, UBII has several communication interfaces, namely Interprocess Communication (IPC), Representational State Transfer (REST) and Sockets. For better understanding all interfaces will be introduced.

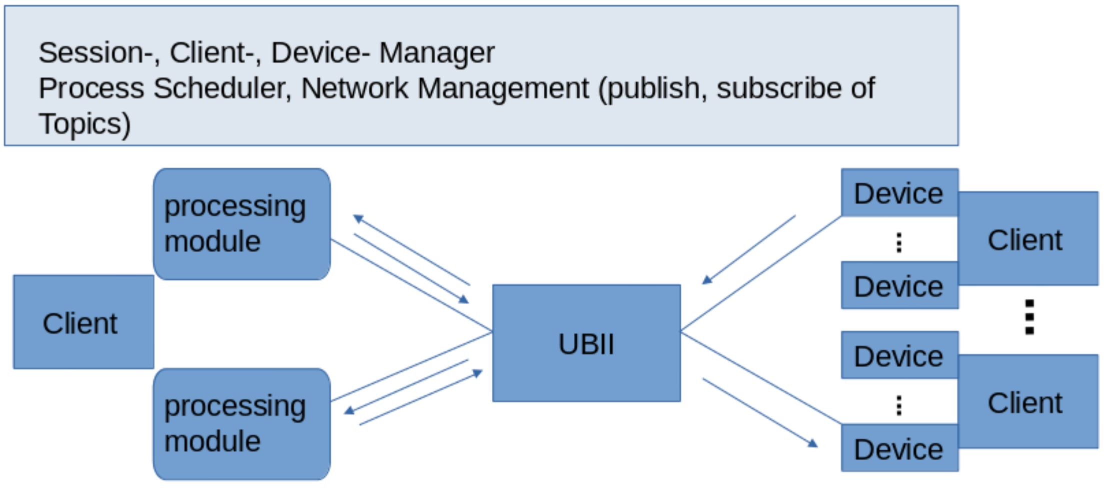

Session-, Client-, Device- Manager
Process Scheduler, Network Management (publish, subscribe of Topics)
processing module
Client
processing module
UBII
Device
Device
Client
Device
Device
Client

Figure 2.1: This picture depicts an abstract example of an UBII network. UBII networks consists of Clients which can have Devices. This Device has components which can either publish topics specified by UBII or subscribe to them. An example flow has been depicted. The Device on right top publish Data and multiple Processing Module subscribed to it and also publish the result of a computation using the data. At the end the device on the right bottom subscribed to that. Processing modules are functions which can process incoming data from a topic. UBII manages the Clients/Devices and Processing modules.

The term IPC refers to the exchange of data between processes. An important aspect in the IPC, is the regulation of the data traffic between the processes, as well as the administration of shared memories. For this purpose, various standards have been developed that have been optimized for use cases such as the following list:

- The use of a shared memory by multiple processes.
- Dependencies of different processes on each other.
- The passing of data among processes.

The operating system SystemV (SysV), which was already published in 1983, dealt among other things with these or similar problems and developed possible solution strategies, which were implemented in various UNIXes as well as UNIX-like systems. [Sec] For UBII it means that if an applications have been written for UBII, but only one client is available, that the distributed application can be executed efficiently on one client, because the communication takes place via IPC.

Although it is possible to run UBII as a distributed application on a single client, this would neglect the advantages of distributed applications. For communication with other devices sockets are used. As already mentioned, it is intended that independent processes, which may only communicate with each other over the internet, comunicate with each other. This means: The IPC strategy to be selected should preferably be able to handle the addressing type IP or the transport protocol TCP/UDP, which is common in the Internet. For this purpose a standardized interface has been developed for common operating systems, whose communication node, a socket, can also be reached via the network. Two programs that want to communicate with each other build a socket for this purpose, which are both addressed with a name. The name is chosen from a valid domain. An example for a valid domain would be AF_INET , which stands for the communication over the internet, so a valid name would be an IP-address with an associated port. It is common for the name to be bound to a socket using a bind which is basically nothing more than an association to a filedescriptor. Further information can be found in [Sec].

As already mentioned, new errors can arise in distributed applications that must be taken into account during implementation. In order to avoid a class of errors and to build up a successful message transfer via sockets with as little knowledge as possible in the area of networking or to fall back on well-defined patterns from software engineering, UBII was developed with the help of ZeroMQ. Particularly noteworthy are three patterns that ZeroMQ provides. One is the Request-Reply pattern. One is the well-known request-reply pattern, in which a sender sends a message to a receiver and then has to wait for a response. The second pattern worth to mention is the router-dealer pattern that makes it possible to run both, the receiving and the sending of the messages asynchronously. The Publish-Subscribe pattern assumes that there are clients offering messages that can be subscribed to by other clients.

There is one area that sockets cannot cover. Many applications today are not written for a specific operating system but for the browser running on the operating system. The reason is that the browser runs on most common operating systems and also the possibilities, like running in the background, software can achieve running on a browser become more and more. The scenario where applications are running on the browser are also abele to profit from UBII, Websockets are used and not explicitly the REST interface because this would not have been efficient enough, since otherwise a connection would have to be established again and again. This, however, reduces the reability. The

WebSocket protocol starts exactly there and allows a bi-directional exchange between a client and a server. [11]

A REST interface was also built for configuration purposes in UBII. UBII has multiple services for different categories. The categories would be: Clients, Devices, Processing Modules, Sessions, where for Processing Modules and Sessions there are again the subcategories Database and Runtime. The difference is that Sessions or Processing Modules can be stored either persistently or temporary. To these categories the common functions Create, Read, Update, Delete (CRUD) were written. Table 2.1

TOPIC	URL	Explanation
SERVER_CONFIG	/services/server_configuration	This will return an Object with the current server configuration.
CLIENT_GET_LIST	/services/client/get_list	A list of all Clients will be returned.
DEVICE_GET_LIST	/services/device/get_list	A list of all Devices will be returned.
PM_DATABASE_SAVE	/services/processing_module/database/save	Persistent save of a Processing Module
PM_DATABASE_DELETE	/services/processing_module/database/delete	Delete Processing Module from DB.
PM_DATABASE_GET	/services/processing_module/database/get	Read Processing Module from DB.
PM_DATABASE_GET_LIST	/services/processing_module/database/get_list	A list of all saved processing modules will be returned.
PM_RUNTIME_ADD	/services/processing_module/runtime/add	Non-persistent save.
PM_RUNTIME_REMOVE	/services/processing_module/runtime/remove	Remove from runtime.
PM_RUNTIME_GET	/services/processing_module/runtime/get	Read from runtime.
PM_RUNTIME_GET_LIST	/services/processing_module/runtime/get_list	A list of all PMs from runtime.
SESSION_DATABASE_SAVE	/services/session/database/save	Save a session in the database.
SESSION_DATABASE_DELETE	/services/session/database/delete	Delete a session in the database.
SESSION_DATABASE_GET	/services/session/database/get	Get a session from the database.
SESSION_DATABASE_GET_LIST	/services/session/database/get_list	A list of all sessions from the database.
SESSION_RUNTIME_ADD	/services/session/runtime/add	Add a new session.
SESSION_RUNTIME_REMOVE	/services/session/runtime/remove	Remove a session from runtime.
SESSION_RUNTIME_GET	/services/session/runtime/get	A list of all sessions from runtime.
SESSION_RUNTIME_GET_LIST	/services/session/runtime/get_list	A list of all running Sessions will be returned.
SESSION_RUNTIME_START	/services/session/runtime/start	Starts a session.
SESSION_RUNTIME_STOP	/services/session/runtime/stop	Stops a session.
SERVICE_LIST	/services/service_list	A list of all Services will be returned.

Table 2.1: This table are explaining most of the services running on UBII.

2.1.3 Mensch-Maschinen-Interaktionen

UBII was mainly designed to send data based on a underlying HCI. HCIs can be understood as a natural consequence that machines or computers have brought with them. The reason for this is that the best machine is only of use if it can be operated efficiently and correctly. An HCI should consist of two categories, functionality and usability. While functionality provides a set of actions, usability provides efficient and adequate interfaces. The effectiveness of an interaction is then the successful balance between these two categories. [S+08] Ubi-Interact was designed to deal among other things with interactions. Interactions can be very complex or simple. In [S+08] the following definitions are made. The complexity depends among other things on the variety of interfaces that an interaction digitizes. The used inputs and outputs through which an interaction passes are also called communication channels. Each independent channel is called a modality. These modalities are divided into 3 categories:

- Visual based interactions.
- Auditory based interactions.
- Sensor based interactions.

Visually based interactions include, for example, determining facial expressions or recognizing the direction of gaze. While the determination of the expression only on the basis of a photo or film of the face is relatively complex, direction of gaze could be realized e.g. with the help of an eye tracker. Roughly speaking, an eye tracker emits infrared light, which is reflected by the eye. The system detects the eyes, e.g. with the help of image recognition algorithms and analyzes the direction of gaze using the reflected pattern. This can then be modeled with a three-dimensional vector $v = (x, y, z)$ with $x, y, z \in R$ The direction of gaze is very useful in different scenarios. For example, if an object standing in space needs to be detected, the selection can be done among other things done with the help of the gaze direction. Furthermore, it allows to analyze the attention of the user or to determine the focus. For Ubi-Interact it makes sense to support a vector data type. Auditory based interactions are interactions based on sounds. Among these are speech recognition, speaker identification or the analysis of emotions also the analysis of emotions based on different acoustic features. In the analysis of the emotions would be e.g. a method that recorded signals are divided in such a way that statistical functions can be used for the evaluation. Recorded signals can be splitted in e.g. fixed time intervals in the recording or already the filtered words. Important indicators can be, depending on the the pitch, the intensity, the speaking rate or also the voice quality. For anger, for example, it has been found that many people have a medium or high pitch in their voice and the intensity or speaking rate is increased. These indicators could also be modeled with a vector or even a boolean.

Sensor-based interactions are probably the most common. The category includes all interactions that are realized with at least one physical sensor. These include, for example, mouse and keyboard, joysticks or but also the common rotation rate and acceleration sensors in cell phones. If one considers e.g. the rotation rate sensor, then the incoming data can be mapped that it describes a rotation from one orientation to the other orientation. The rotation around an arbitrary axis, can be modeled in the two-dimensional e.g. with the help of a matrix $M \in R^{2x2}$. The dimension of the matrix increases to 4x4, if homogeneous coordinates are used. This model bears the danger that if two axes rotated in such a way that they are parallel to each other, the degree of freedom about one axis is lost. As a remedy, the rotation can be expressed with a quaternion. A quaternion is a number $q = (s, v) = (s, x, y, z) = s + xi + yj + zk$ with $s, x, y, z \in R, v \in R^3$ and i, j, k are three imaginary units. A rotation of a vector can be done after the transformation into a quaternion, then a defined multiplication can be performed resulting in a new quaternion which then can be mapped again to a vector. UBII supports these data types already. The Equation 2.1 shows the corresponding matrices for the rotation around α in the given model around an arbitrary unit vector $\vec{n}$.

$$\begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix} \begin{pmatrix} n_1^2(1-cos\alpha)+cos\alpha & n_1n_2(1-cos\alpha)-n_3sin\alpha & n_1n_3(1-cos\alpha)+n_2sin\alpha & 0 \\ n_2n_1(1-cos\alpha)+n_3sin\alpha & n_2^2(1-cos\alpha)+cos\alpha & n_2n_3(1-cos\alpha)+n_1sin\alpha & 0 \\ n_3n_1(1-cos\alpha)+n_2sin\alpha & n_3n_2(1-cos\alpha)+n_1sin\alpha & n_3^2(1-cos\alpha)+cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos(\alpha/2) \\ n_1sin(\alpha/2)i \\ n_2sin(\alpha/2)j \\ n_3sin(\alpha/2)k \end{pmatrix} \tag{2.1}$$

2.1.4 Topics

The last topic relevant for this work are components of Devices. As mentioned, clients can register to an UBII network with their devices. After that, devices can either publish TOPICs or subscribe to them. Topics are either simple data types like Boolean, Float, etc. or data structures like lists, maps, etc. or special data types for example used by certain devices like Myo. In order for the data to be transferred relatively efficiently, it is serialized or deserialized using Protobuf. Google Protobuf is Google's equivalent to XML is a markup language developed by Google, where data can be structured and then translated into different programming languages with the help of the protoc compiler. The translated classes are platform independent and offer among other things Methods for serialization. [Buf]

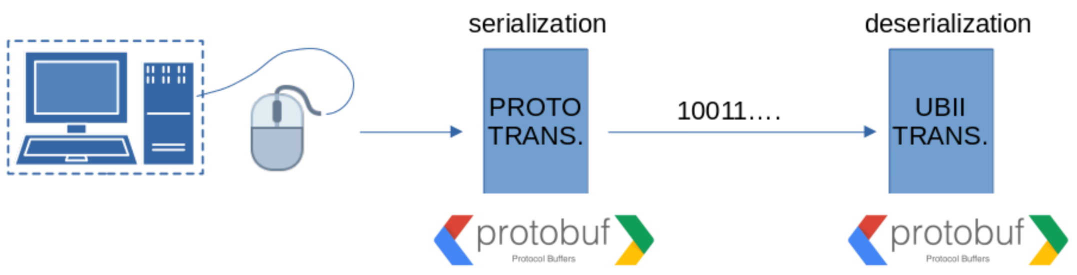


Figure 2.2: This pictures shows the the Process of msg. translation. A Computer with a mouse as Device sends for example the events of the mouseclick as boolean to the UBII Network for further processing. Every time the Data has to be encoded or deencoded done by protobuf. TRANS. stands for translation. Protobuf logo taken from https://www.freecodecamp.org/news/googles-protocol-buffers-in-python/

2.2 Graphs

Graphs are essential in computer science and therefore it will only be briefly discussed and short definition is given. A graph consists of a set of nodes $v \in V$ and edges $e \in E$. Nodes can be connect to other nodes using one or more edges $(u, v) \in E$ with $u, v \in V$. Graph can be categorized in either directed and undirected graphs. This means that edges only have one direction. For this work, only directed graphs are of interest since the publish/subsrcibe pattern is uni-directional. If $tail(e)$ and $head(e)$ is equal to v it describes a self loop.

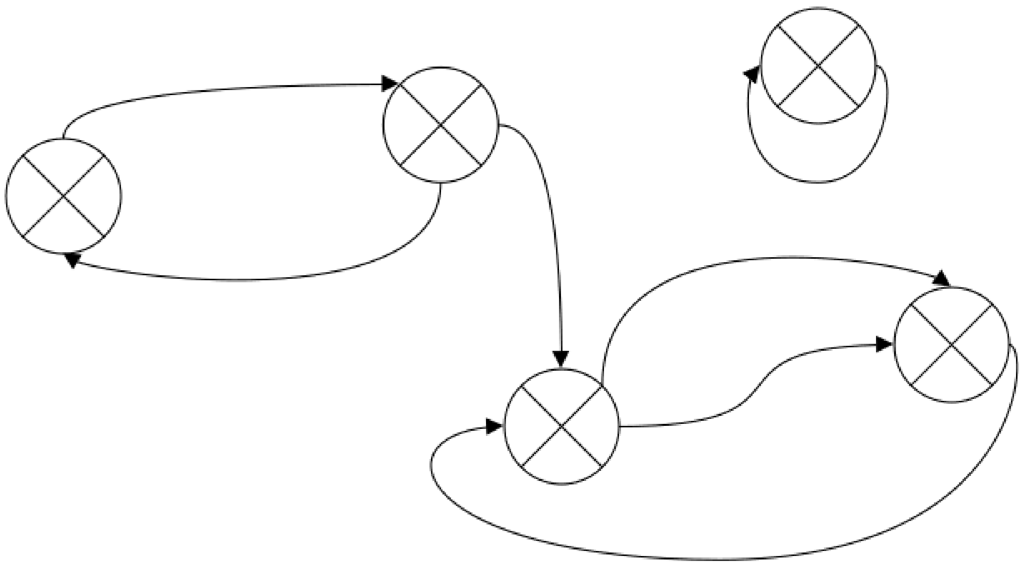


Figure 2.3: A graph with a cycle and directed edges and 3 different strongly connected components. On the top right is a node with a self loop.

Graphs can be used for a wide variety of applications since it can be connected to a set of problems appearing in the real world or in mathematical models. In order to connect a graph with a real world scenario, a mapping must be found that transfers the problem specification to a graph. A common example is the path of the traveling salesman. Here cities are represented as vertices and paths to these cities as edges. In this example costs are assigned to edges, e.g. time. With the help of these mappings, questions like "Find the shortest path from A to B" or similar questions can be answered efficiently. Even if a graph structure is not directly needed, in order to apply well known algorithms, in some cases it is simply easier to represent a problem set as a graph, so the configuration can be done on the graph instead of filling out forms. In [Igu] three advantages are mentioned, why a visualization of a graph makes sense:

- To find something
- To illustrate something
- To discover features

Regarding the first point, despite metrics and statistics over a certain dataset, it may be that certain structures are not revealed. A graph might be able to reveal such structures.

As an example, a graph can be taken where vertices represent persons and an edge represents the relation: 'u knows v' with $u, v \in V$. Families could be characterized by finding a group of fully meshed vertices.

The second point is relatively logical, since it has been shown for decades that graphs are very good at representing complex structures, either to impress or to capture something. Especially with UBII, a graph would solidify UBII's concept.

The third point is probably the most interesting. As indicated in Figure 2.3, there are several strongly connected components in this graph. Strong components can be found in different ways. In the first point the fully meshed network was mentioned, which could be a strongly connected component. Mathematically spoken using the valence and the n-ring as metric of a group of nodes. The valence gives for each node the number of from- and to- edges and the n-ring considers a group of nodes to which there is a directed or undirected path, which runs over max $n \in |N$ nodes. So in the example the 1-ring would be of interest and a valence equal to the number of the nodes in the 1-ring.

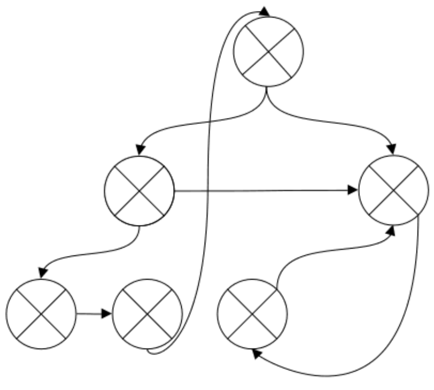
Another observation would be, if nodes are considered as coordinates and close coordinates are defined as a strongly connected components, finding this groups in a set of coordinates can be done with k-means. Such considerations then imply features which can be helpful for the respective problem. In [Igu] s.o. can read more about advantages and problems of graph layouts.

A graph can therefore be useful to discover strongly connected components that data sets can bring with them. Finding features and more, it is necessary that nodes in the graph have to be positioned in such a way that features are getting visible and human readable. For this it can help to use a layout. There are many layouts that make sense for different situations and improve the readability of a graph. One category would be topologically generated graph layouts. The simplest example is a tree which can be created with a depth-first search (DFS) or breadth first search (BFS) and a step wise arrangement of the nodes in the corresponding level. An other example could be a linked list where nodes are arranged in a line or a grid which can be created by simply specifying the dimension as a parameter and then arranging the nodes in equidistant distances in the horizontal direction if the dimension is one or additionally in the vertical direction if the dimension is 2. If the number of nodes exceeds a threshold in row a break can done so that the next node will be visualized in the next row.

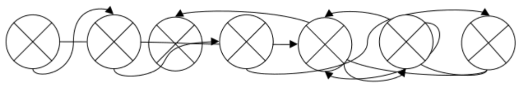
Another kind of layout would be a force-directed graph. The next paragraph will only cover the basics. Such graph layouts can be generated using various mostly physical laws. One possibility would be to discretize each node with a mass m. The forces between the nodes can be described by F_{ij}^{int} and the force on a node by $F_i = F_{i,j}^{int}$. *int* means internal forces. With the conditions that an action equals a reaction, it then follows that: $F_{ij}^{int} = -F_{ji}^{int}$, $\sum_i \sum_j F_{ij}^{int} = 0$ must hold. With the help of Newton's second law an acceleration a_i can be derived and then a position or velocity by integration. The question is now how to model the internal force. An example would be Hook's Law which can be defined with the help of the stiffness k, the initial length L and the current length l, namely: $F_{ij} = -k(l-L)\frac{x_i - x_j}{l}$. There also other formulations like using potential fields or gravity. To make the system stop, a damping $-\gamma\frac{dx}{dt}$ with $\gamma \in [0,1]$ can be introduced. Now a formula for motion can be derived:

$$m_i \frac{d^2 x_i(t)}{dt^2} = F_i^{int}(x_i(t)) - \gamma \frac{dx_i(t)}{dt} \tag{2.2}$$

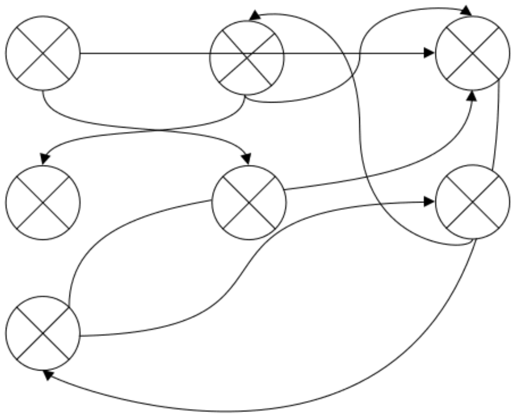
This is an ODE of second order and can be solved by using integration, for complex graphs this requires numerical integration as they are too complex for an analytical solution. Unfortunately, the disadvantage is that for large graphs it takes time to calculate the positions of all nodes if a simulation of this type is used. But the advantage is that for a large set of graphs it can give a visual advantage (see 3 points above). More about force-directed graphs can be read here. [Mil10]



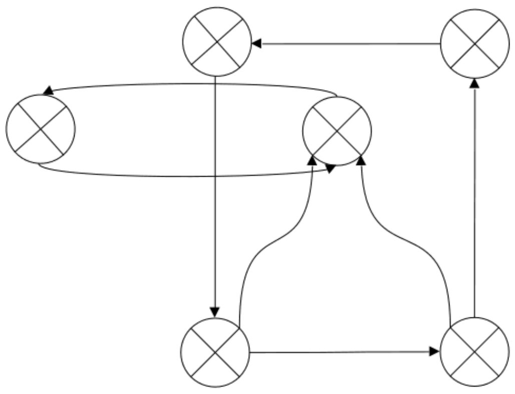
(a) The Layout in this case is a tree. For this particular Graph this Graph looks clear, because there almost not intersections of edges, but this is because of the root was chosen in the right manner. If an other root would have been chosen the relationship between nodes could rely another structure.



(b) The Graph is very compact but the relations between nodes are not easy to understand. To improve this graph one could reorder the nodes and bend the edges based on the distance and making the edges larger.



(c) The Graph looks more clear than the list. But no structure can be implied.



(d) There is a structure in the graph and it is almost symmetric. The intersections of edges are 3 making it readable. The structure was created with `https://console.neo4j.org/` which represents Graphs as force Graphs and can be created with Listing 2.1

Figure 2.4: This figures shows the same Graph four times with a different Layout. It is clear to say that one is more suitable to represent the Graph for the three listed reasons than others for several reasons.

```
CREATE (Neo:Crew {name:'Neo'}), (Morpheus:Crew {name: 'Morpheus'}),
(Trinity:Crew {name: 'Trinity'}), (Cypher:Crew:Matrix {name: 'Cypher'}),
(Smith:Matrix {name: 'Agent Smith'}), (Architect:Matrix {name:'The Architect'}),
(Neo)-[:KNOWS]->(Morpheus), (Neo)-[:KNOWS]->(Trinity), (Morpheus)-[:KNOWS]->(Trinity), (
    Morpheus)-[:KNOWS]->(Cypher), (Cypher)-[:KNOWS]->(Smith), (Smith)-[:KNOWS]->(Neo), (
    Architect)-[:KNOWS]->(Trinity), (Trinity)-[:KNOWS]->(Architect)
```

Listing 2.1: This cypher creates the Graph in Figure 2.4 for Neo4j as an example on how a force graph visualize such small graphs.

2.3 Debugging Applications and Distributed Applications

2.3.1 Debugging Applications

Debugging is a very general term that needs to be defined. In [Agi02] certain rules are stated like:

- understand the system
- be able to generate errors
- be able to see errors instead of just thinking about them
- Divide and Conquer
- incremental change
- be able to log the system

The following can be said about the points. First of all the system has to be understood completely, which means that for each input it has to be derivable which output has to be generated, thereby it should be understood which task one or more functions have.

Next, one should be able to generate errors, this helps for example if an error class is suspected to contain an error, one should be able to prove it with the help of an intentional error within this class that actually produces an error.

It is important that you can see these errors, i.e. that there are outputs that make these errors visible, so s.o. can be sure.

It is also important that one can divide the system into several parts, so that one can test only special classes, without having to test the whole system.

This also leads automatically to the fact that there should be a possibility for the gradual change.

Another important part is that a time sequence can be theoretically created of all outputs so that it is possible to see in retrospect when certain outputs or errors occurred.

There are several tools for development environments and most of them have a similarity. On the one hand it can be selected in the development environment whether the program should be debugged or not. Furthermore it is common that there is both a play button and a stop button. Play stands for beginning the debugging process. Another button is forward button which is able to skip a line of code or a whole segment. Furthermore, a breakpoint can be selected that stops the runtime if the line in question has been reached. If the execution of the program reaches this breakpoint, it is possible to examine the state of all possible variables of the program with the help of an inspector. These are the minimum requirements for a debugger introduced in most tools.

2.3.2 Debugging Distributed Applications

In distributed systems, the application is distributed on heterogeneous systems. This also means that more bugs can occur than in an applications that runs on only one system and was written by only one developer. Therefore the following points must be considered [Cou+12]:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers.

An application can run on a client that lives in a completely different network topology than another client that needed for the application to work as a whole. Applications can run on different operating systems and support different hardware. Furthermore they can be realized with different programming languages and last but not least they can be developed by different developers. Furthermore, these operations usually run in competition with each other and the results may have to be combined. Also all of them can have a local time and a global time is hard to derive.

2.4 Web Applications

Web applications are applications that are executed with the help of the browser. The advantage is that web applications run on any operating system that provides a browser. The disadvantage is that the application is dependent on the web browser and its functionalities. Thus, in some cases it is not possible or disadvantageous to write an application only for the browser. The standard language for web applications is JavaScript, but various derivatives have already been developed, such as Angular, React or VueJs. For UBII there is already an editor written in VueJs which is why this work is written in VueJs together with Bootstrap, a framework for Cascading Style Sheets (CSS). Unfortunately different browsers use different compilers to execute a Hypertext Markup Language (HTML) page which is why this work is optimized for Chrome.

Beside HTML two other important technologies were used. First canvas, in canvas it is possible to display graphics without the need of a Document Object Model (DOM). This offers particularly performance advantages because no DOM has to be created in order to draw or display elements but thereby it lacks a lot of functionalities that HTML would provide. Another important technology used and discussed in this thesis are the different storage possibilities of the browser. Data can be stored in a browser in multiple ways and either temporarily or persistently. On the one hand there is localstorage, IndexDB or Cookies. The difference between localstorage and indexdb are the capacities. In contrast of the two, cookies can also be sent to the web server. On the other hand there is a session storage with a lifcycle connected to a session.

3 Related Work

In this chapter, related work and recent scientific findings are discussed. First, different applications are shown in which graphs are used today. Then different layouts are presented.After that different editors are presented which provide graph drawing on the web. At the end some relevant work is shown how debugging of distributed applications can be achieved.

3.1 Applications using Graphs

Today, graphs are used in a wide variety of applications. Thereby it is mostly distinguished for which area one wants to use a graph.

Graph drawing is relevant to much of the social sciences but its most direct association is with social structure and social relations. The analytic concept of social networks has been linked so closely with its representation as a graph that the use of related graph-theoretic techniques in any discipline is often considered an application of social network analysis. [Bin+19]

Social networks in the strict sense consist of actors and the social ties that moderate their actions. Variant types include affiliation networks (depending on context, represented as hypergraphs or bipartite graphs with a fixed bipartition), ego networks (represented with or without the defining focal actor who is in relationship with everyone else, and with or without relationships between the other actors), and longitudinal networks (given, for instance, as cross-sectional panel data, interval-censored aggregations, or relational events). Descriptive features include macro-level classifications such as being a core-periphery or small-world type network as well as structural properties such as cohesive groups, roles, and actor centralities. Statistical inference is often based on particular families of models for which there is a long history. [Bin+19]

Network graphs are used to display the structre of a network created by computers. In this diagram nodes are clients with ip adresses and edges are routes to other clients in the same network or over a router to other networks. For network graphs for example [CBB] collected and recorded routing paths from a test to each of over 90,000 registered networks on the Internet since August 1998, resulting in a reachability database. In order to layout the graph a simulated spring-force algorithm was used.

Model-based design (MBD), also referred to as model-based development or model-driven engineering, is a design methodology where some artefact, referred to as system under development (SUD), is created based on some model(s) of it. This model (or collection of models) is initially rather abstract, concentrating on what the SUD is supposed to do, and only in later—possibly automated—development stages it is specified how the SUD does what it does. [Bin+19]

To represent UBII two different application areas were potentially available, on the one hand UBII could be represented as a network graph or a model based graph. Since it is not the task to find out how clients, devices and processing modules are distributed in the network it was not decided to represent it as a flow chart. The reason is that processing modules can be understood as functions and thus have a control flow.

3.2 Layout effects

In the basics it was mentioned that the layout is crucial and three criteria were mentioned when a layout can be considered useful. In [PRM95] this was again supported with a study. It is argued that a graph should help the user to understand the structure or even give new information and also help to remember certain information. Furthermore, the author believes that a good layout can say more than a picture with a thousand words and a bad layout can do the opposite and even irritate the user. Furthermore, the author confirms that there are measurable aesthetic qualities that can classify a graph. The author names the following qualities:

- bends: if a connection between two nodes exists but is blocked by other nodes, the edge would theoretically have to bend to avoid them.
- crossings: number of arc crossing should be minimized
- symmetry: where possible a symmetric view should be displayed
- orthogonality: arcs or arcs segments should be parallel to the coordinate axes
- minimum angle: the minimum angle formed by the drawing of consecutive arcs around a node should be maximised

These theses were tried to be validated in [PRM95], three hypotheses were made and the first two were confirmed:

That as the number of inflections in the graph is increased the understandably is reduced

The more the edges intersect the more the understandably is reduced

The only thing that could not be confirmed was symmetric local groups increase understandably

In [HHE06] there is a comparison of the relative effectiveness of five sociogram drawing conventions in communicating underlying network substance, based on user task performance and usability preference, in order to examine effects of different spatial layout formats on human sociogram perception. The authors also conclude that edge crossings should be avoided. Some of the tested Layout are shown in Figure 3.1 Even though this layouts are most used in sociagram it is also interesting to try this layout with other applications.

3.3 Graph drawing for the Web

For the web there are some interesting graph libraries which are open source and feature rich. One work worth to mention is rete.js [ret]. Unlike other it allows to create node based directly in the browser using HTML and CSS. Figure 3.2 The andvantage is that is easy to integrate a lots GUI written in HTML, Js and Css. Another canditate drawflow [dra] can be used to generate flow charts. Figure 3.3 It already comes with features like dragging nodes, multiple connections, Zoom in/out and more. Last but not least comes litegraph.js [lit] which also comes with a lot of features, but also with a demo which provides a lot of examples, which others didn't provide. Figure 3.4

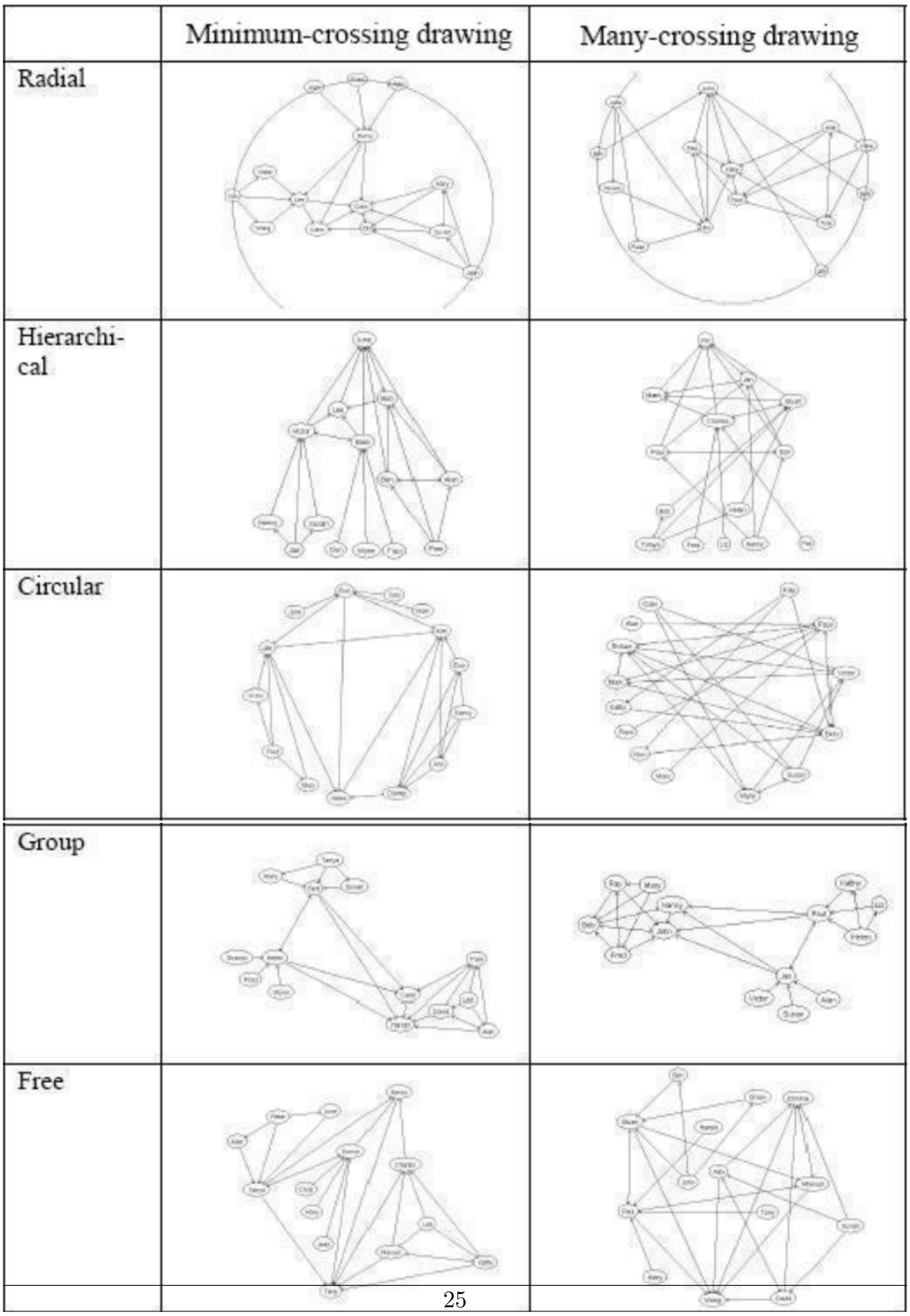


Figure 3.1: Some Layouts for a graph. Figure taken from [HHE06]

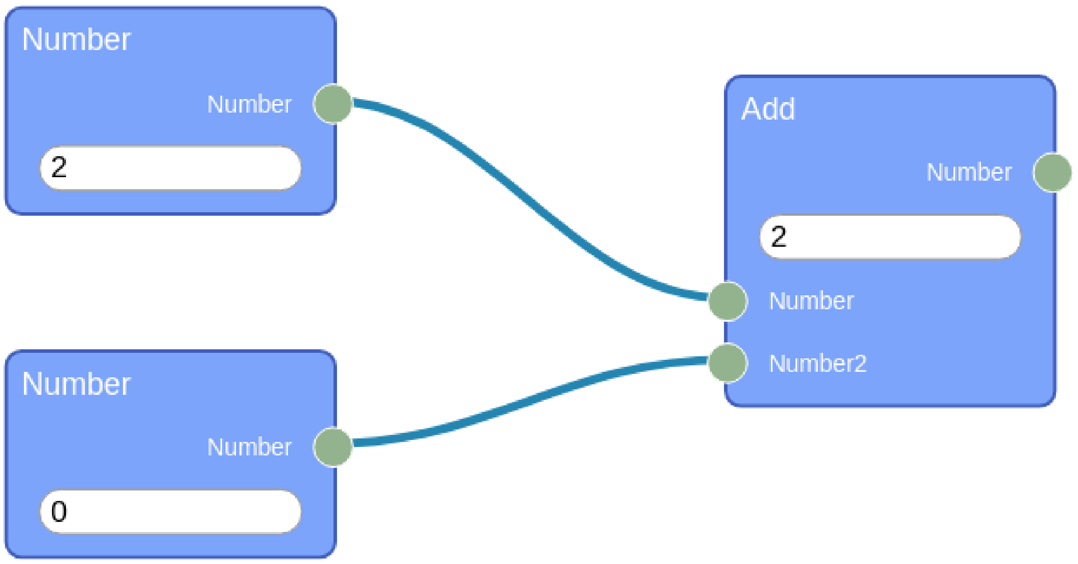


Figure 3.2: An example Graph written inRete.js. Figure taken from [ret]

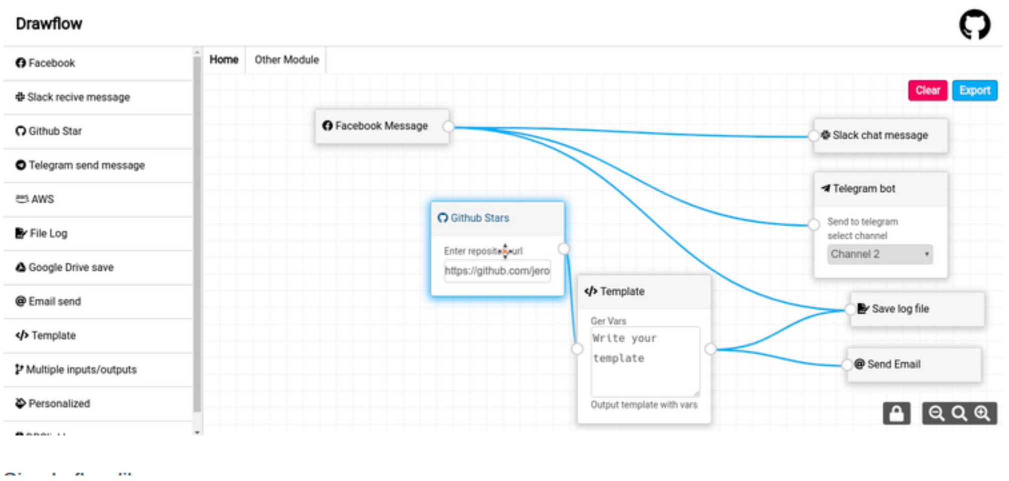


Figure 3.3: An example Graph written in drawflow. Figure taken from [dra]

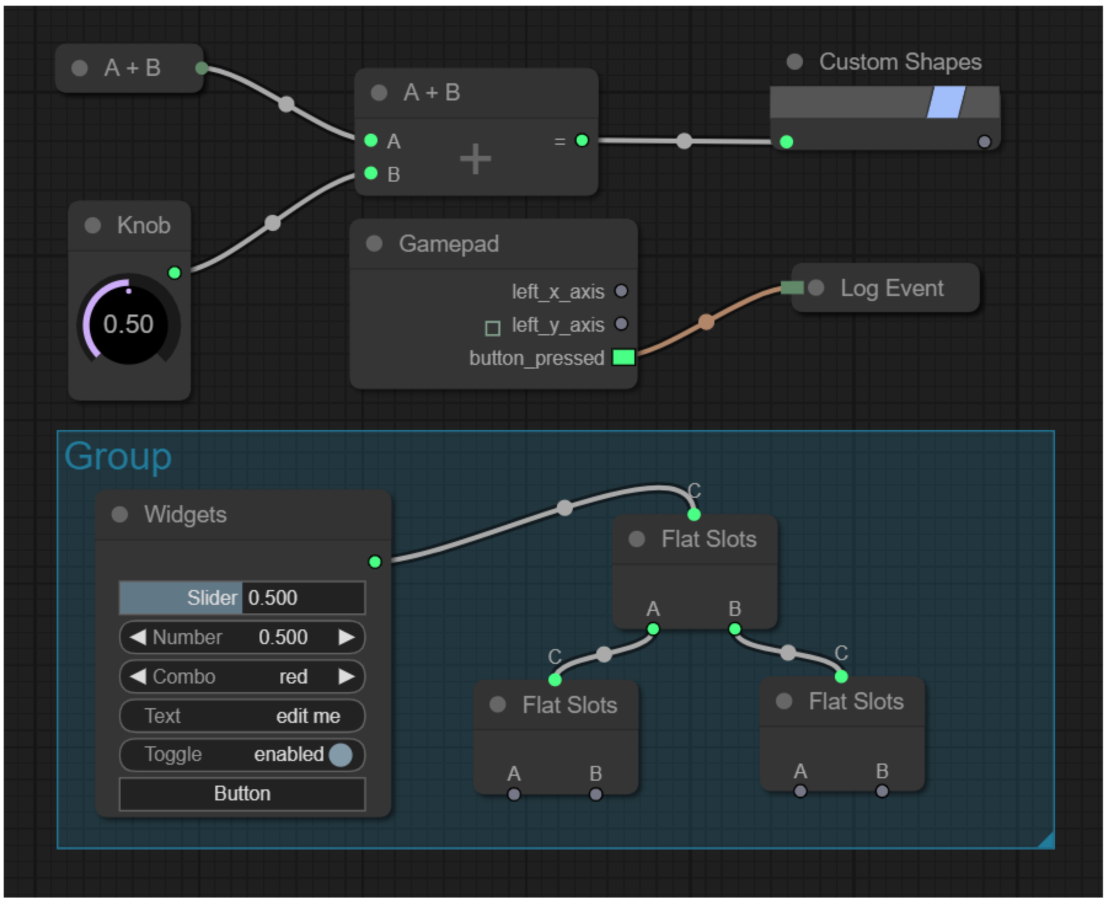


Figure 3.4: An example Graph written in litegraph.js. Figure taken from [lit]

3.4 Debugging Distributed Applications

In [Cou+12] it is stated that there is problem recording a system's global state so that important statements about the temporarily state can be done. In general debugging a distributed system is just that. The author says that an example would be a system of pipes in a factory where s.o. is interested in whether all the valves, which are controlled by different processes were open at some time. Here it is not possible to observe all the values of the variables simultaneously and the challenge is to monitor the systems execution over time, so that in can be inferred post hoc whether the required safety condition was o may have been violated. Further it is explained how the state can be collected from the processes by just sending an initial state and then only the changes of the variables in order to monitor the system. This could also be done for UBII.

4 Implementation of the Graphical User Interface

In the next chapter the implementation will be discussed. First it will be explained why litegraph was chosen. Then it is explained how UBII can be displayed with the help of a graph. Finally it will be explained in detail how the components of the graph can be used to configure UBII. Finally, the implementation is discussed. First the hardware and libraries are presented, then the important algorithms and data structures are presented and finally a suggestion is made how the graph can be created. Finally, it is tested how well the implementation is performing in certain situations and which criteria have hold for specific tests.

4.1 Approach

4.1.1 Graphs on the Browser

Unlike e.g. tables or other HTML elements there is no graph element for the browsers yet. The first question that was asked was whether a custom graph should be implemented or an existing library should be used for it. A self-implemented solution would have the advantage that it could be optimized to the problem, but the disadvantage that this would cost a lot of time, so it was decided against it. Therefore it was first researched which open source alternatives or solutions exist. A critical decision point was which language to use to create a graph. There were three different possibilities:

- HTML
- Canvas
- Scalable Vector Graphics (SVG)

HTML would have several advantages, such as relying on standardized input/output masks that have already been implemented for both UBII and common HCI. But it is not very performant to implement a graph with many nodes in HTML.

One use case for UBII could be that it is used for a lot of Clients, Devices and Processing

Modules i.e. UBII is meant to represent large networks as well This means that HTML could be Problem. Therefore, only the other two technologies were left. Both SVG and Canvas are very performant, but it must be noted that with SVG a tree of elements must also be created, while with Canvas the primitives can be rendered directly, which is why Canvas indeed requires more effort to realize a GUI, but brings a performance advantage, espacially when many primitives needs to be drawn. For completeness, WebGL outperforms both technologies. If the device is able to interpret WebGL. Here it must also be distinguished whether simple WebGL is used or complex algorithms optimized for WEBGL are used.

4.1.2 Mapping of UBII

As already mentioned in the basics, UBII is a framework for distributed applications in a network. The first question that has been raised is whether UBII could be configured using only input and output masks. While it is possible, it would be a realtive complex user interface and relatively unnatural and confusing, since clients, devices and processing modules have a lot of information. It would also require an explanation of how to use forms rather than the UI explaining the system as well. Therefore, it was decided quite early that UBII should be configured using a graph. To make this possible, a mapping must first be found, i.e. a mathematical description of UBII, as defined in the basics on graphs. To do this, the first point is that UBII consists of Clients, Devices, Components, and Possessing Modules. Furthermore, it must be stated that a client can host multiple devices and the devices can have multiple components. A component is being used for inputs and outputs identificated by variables. And one inpurt or output of the devices can subscribe or publicsh one or more topics as indicated in the basics.

The first mapping is described in Figure 4.1. In this mapping devices, processing modules and components are considered as nodes that have a relationship to each other. There are two to three types of relationships or edges, one of them would be the typical "has" relationship and for the other two, it could be considered as a publish or subscribe relationship or even combination of them. The advantage would be that it is relatively obvious that for each class of UBII there is also a node, but if this graph were used for a relatively large distributed system, the graph could very easily degenerate. The typical example would be a PC as a client with a mouse and the topics Coordinates as Vector2D and possibly boolean for the mouse click, then already four nodes would exist. Furthermore, there would be two groups of edges, which could should be illustrated in the graph, for example, by coloring.

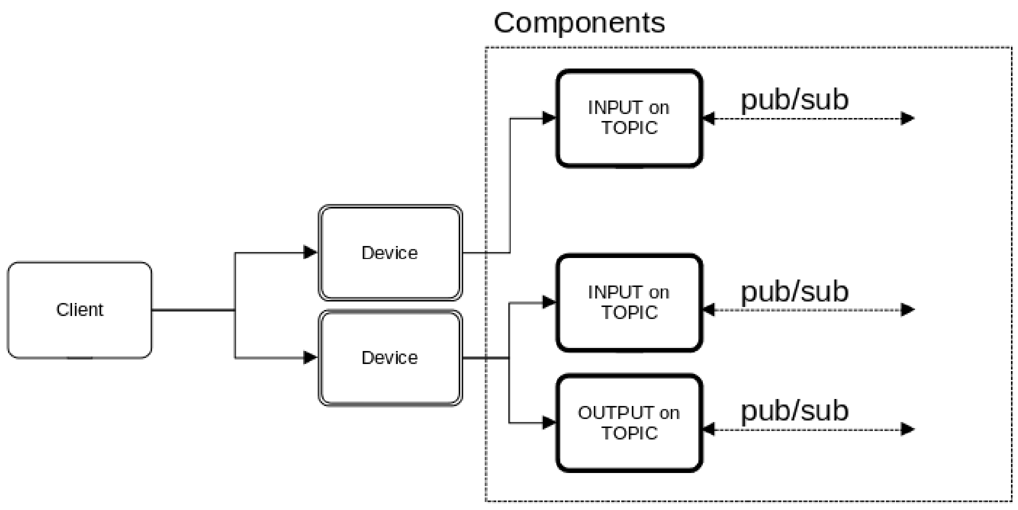


Figure 4.1: This picture show the possibility to map UBII as Graph. In this case Clients, Devices and Components are represented as Nodes and two types of directed edges, namely "has" and "pub/sub", are used for the relationships.

Because of the disadvantages mentioned, it was therefore decided to use a different solution. Instead of mapping all classes with their own node, clients and devices were combined. Since UBII does not use a merged class of these two classes it needs a possibility to identify the client and the device fro the suggested node, therefore a concatenated id was created, which consists of the unique Client and Device name.

```
ID: ${ClientName}.${DeviceName}
```

In addition, there are no longer components that are represented as nodes, but are now connectors that reside within a node. If the connector is on the right side, it is a publish and if the connector is on the left side, it is a subscribe. Just like in the previous solution, Processing Modules are again represented as nodes and also using the connectors.

The advantage is that it simplifies the graph. If, for example, a client has several devices, the number of nodes would be reduced by one, because it is no longer displayed as a node as in the previous solution. Furthermore, components are eliminated because they are now represented as connectors within a node. The disadvantage is that it has become more difficult to assign Devices to a Client, if several Clients exist. The only difference between them is the name of the client. Figure 4.2 Furthermore, the types of edges have multiplied. For each topic in UBII there is now one edge of the respective type. In the example with the PC and the mouse, one node would represent the mouse, which would have two connectors, one publishing a Boolean for the mouse click and the other publishing the coordinates using a Vector2D.

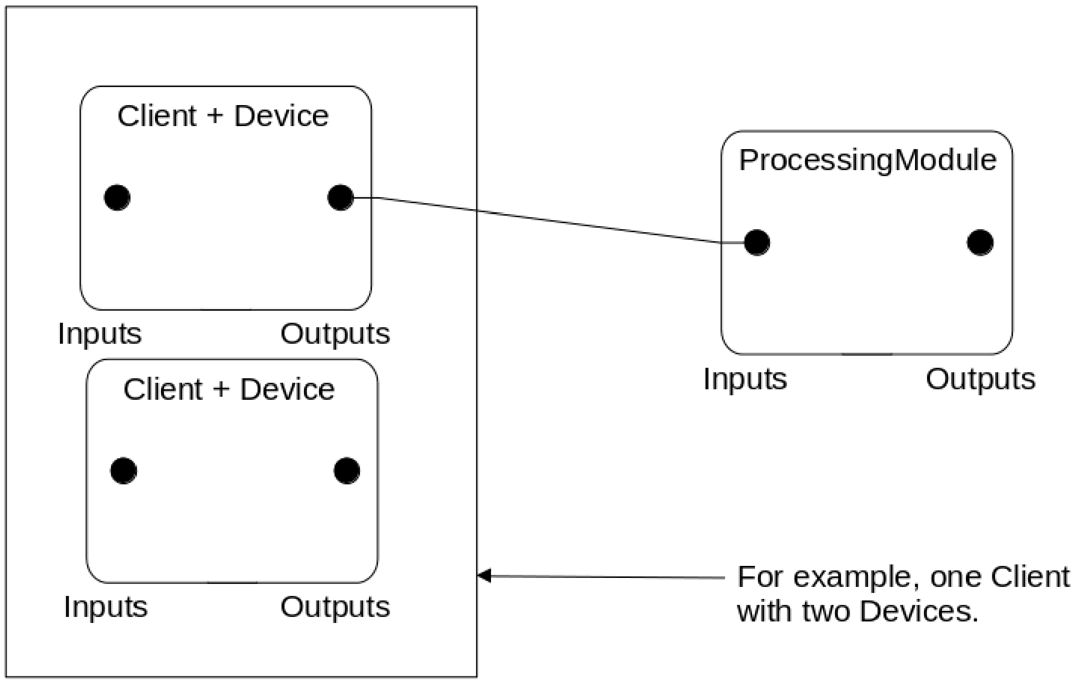
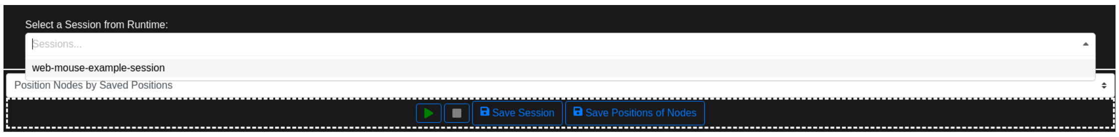


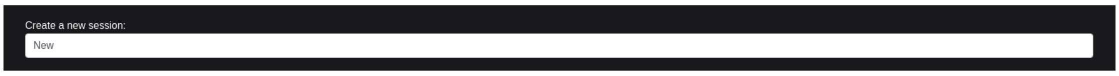
Figure 4.2: This picture show the preferred solution on how to map UBII as Graph. In this case Clients and Devices are represented as Nodes, as well as Processing Modules and there exist multiple types of edges, namely as much as Topics exists, also the position of the black connectors decides if it is an publish or an subscribe on that Topic.

4.1.3 Create or Modify a Session

One of the most important aspects of this work was that UBII can be visualized as Graph and the ability to configure sessions, this means first it requires a selector that allows a selection of sessions. In UBII a session contains all information about the running clients, devices and processing modules as well as the relationships (topics, publish/subscribe) and details about functions. In UBII sessions can either exist at runtime or they are stored in a database. A requirement was also that sessions can be created. To achieve this, it was handled so that if no session is selected, it is handled as a new session and if a session is selected, it is an existing session. A new session needs a name while an existing session already has an existing name. The user should know in advance if he wants to create or edit a session and can change the categories in the tab. Tabs are a way of displaying more information using less window real estate. If the selection is deselected it will reset the complete graph or UI. For ease of use, it was deliberately decided to not create buttons that could be used to create or discard sessions. Figure 4.3


Select a Session from Runtime:
Sessions...
web-mouse-example-session
Position Nodes by Saved Positions
Save Session
Save Positions of Nodes

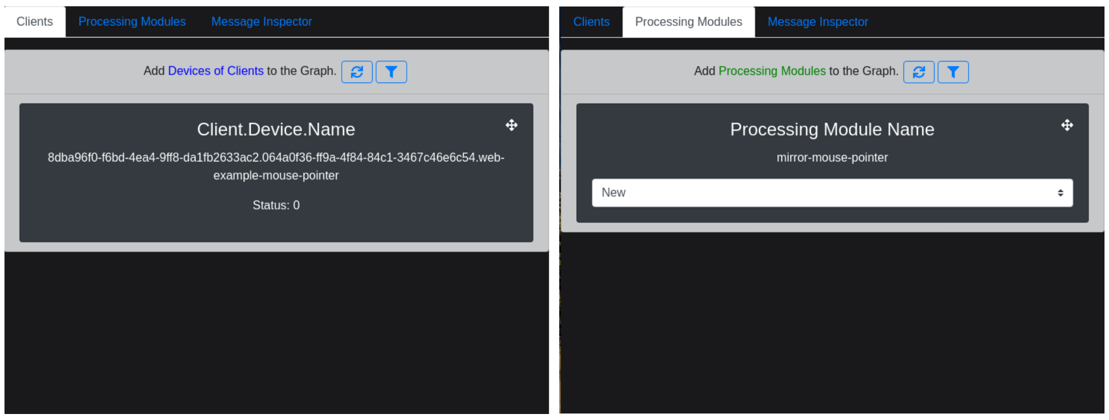
(a) Selection of a Session.



(b) Creating a new Session.

Figure 4.3: The two input masks for selecting or creating a Session.

Now there are two different possibilities, if a session is selected all clients, devices and processing modules are loaded automatically and those that belong to the session are filtered and displayed in a graph. If a new session has to be created, there are two separate buttons that load all clients and devices from the database and all processing modules. As was determined during the mapping, Client and Devices are one category and Processing Modules are another category, so a tab was used to separate them as well. So that these can be identified the name is represented, as well as IDs. Furthermore the status is shown, because Client/Devices or Processing Modules can be active, inactive or unreachable. In order to distinguish between client/devices and processing modules, a color was added: blue was chosen for clients/devices and green for processing modules. Figure 4.4


Clients
Processing Modules
Message Inspector
Add Devices of Clients to the Graph.
Client.Device.Name
8dba96f0-f6bd-4ea4-9ff8-da1fb2633ac2.064a0f36-ff9a-4f84-84c1-3467c46e6c54.web-example-mouse-pointer
Status: 0
Clients
Processing Modules
Message Inspector
Add Processing Modules to the Graph.
Processing Module Name
mirror-mouse-pointer
New

(a) List of Client/Devices with the refresh button to get a current list of them.

(b) List of Processing Modules with the refresh button to get a current list of them.

Figure 4.4: Lists of Clients/Devices and Processing Modules.

There is another special feature, clients and devices can not be configured or created, but must first register with UBII. Processing Modules can be configured, however, for each Processing Module there is a specification, on the basis of this specification an instance can be created. Instances must be executed on clients. Furthermore they have a clock frequency. The frequency specifies how many processing steps should be executed per second. There are still other configuration possibilities like e.g. the language in which the process was written or events, however these were not considered in this work. Also it is necessery to provide a function or callback, that will be executed if data is recieved.

After all clients and devices and processing modules of a session have been loaded or all clients and devices and processing modules have been stored in the database, the configuration in the graph can be started. The possibilities in the graph are particularly suitable for changes of the publish-subscribe of topics. With the help of the selected, already described structure in the graph, relations can now be created or removed. Clients and Devices or Processing Modules can also be removed from the session using the right mouse button. The connectors are all marked with a unique name. To establish a relationship to the clients and devices to processing modules in the list, the color of the nodes has been changed to blue or green. Furthermore the name and the ID for Client/Devices and the name and the ID of the Processing Modules are displayed. For Processing Moldules a slider for the clock frequency is also displayed, as well as a button that opens a modal, in which it is possible to select on which client the Processing Module should be executed. Figure 4.5

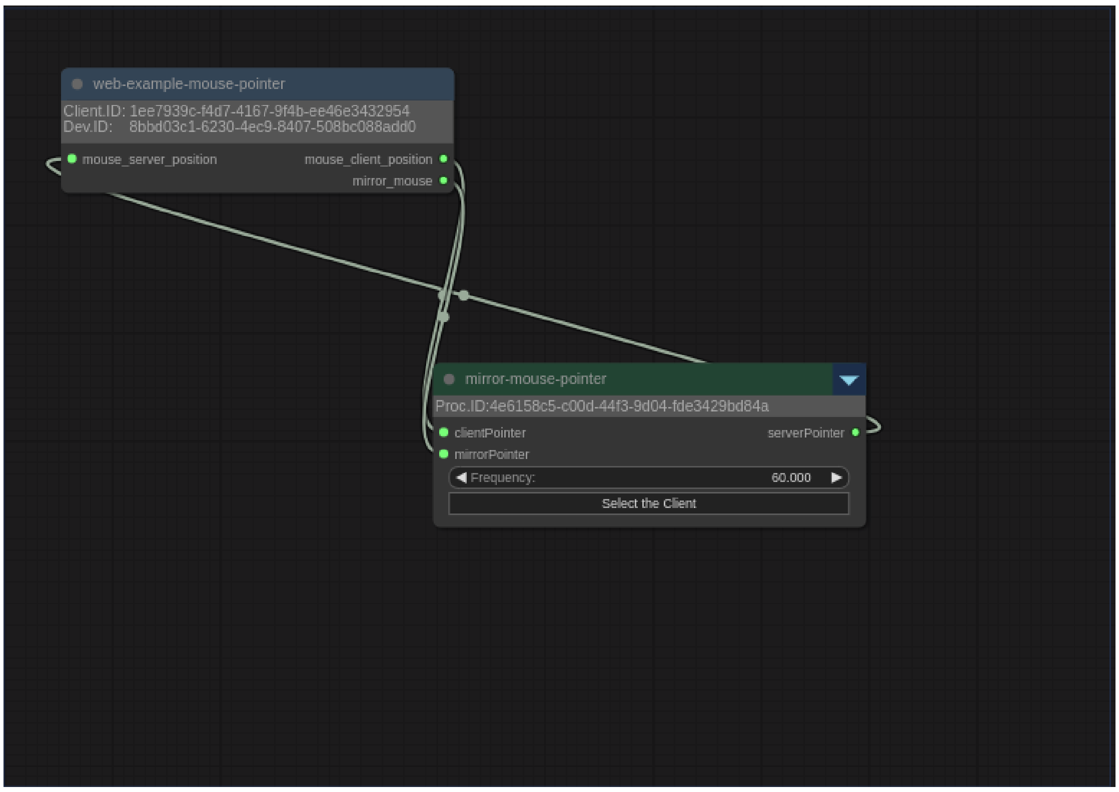

web-example-mouse-pointer
Client.ID: 1ee7939c-f4d7-4167-9f4b-ee46e3432954
Dev.ID: 8bbd03c1-6230-4ec9-8407-508bc088add0
mouse_server_position
mouse_client_position
mirror_mouse
mirror-mouse-pointer
Proc.ID:4e6158c5-c00d-44f3-9d04-fde3429bd84a
clientPointer
serverPointer
mirrorPointer
Frequency:
60.000
Select the Client

Figure 4.5: This is how a session of UBII will be represented as graph. It is possible to see the blue and the green labels. The relationships, edges are also set. Processing Modules also have a slider and button.

The last panel that has been developed is a configuration menu for flow control which consists of buttons with a play and stop icon. The play button will visualize the flow and enable the message inspector as well as the lantency. The flow signalise that an application is running and also the direction of the data sent between two nodes. For example one node publish a boolean and another node subscribed to it. Furthermore, two buttons were added, one is for the layout of the graph which can be locally saved or the settings of the sessions which can be saved in the database of UBII. Figure 4.6

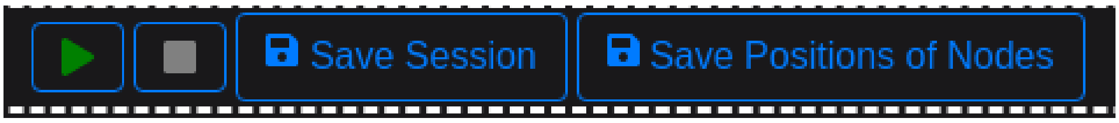


Figure 4.6: This pictures shows the available buttons for configuration and visualizing the flow.

As mentioned UBII will eventually be used for larger networks which means that there could be a lot of nodes. In order to keep the graph manageable, different layout strategies have been developed that can be selected using a selection. In this work, three different layouts were used. First, the positions of the nodes can be stored. Secondly a list or grid of nodes can be generated. And third an arrangement using forces can be selected. All methods have advantages and disadvantages.

For example one could choose to save the nodes locally with a specific layout. For large graphs this could be laborious. Therefore it makes probably sense to first order the graph with a layout which can be calculated quickly like the arrangement litegraph.js is offering. This layout does not visualize a structure or clusters. That's why the force graph can be selected to layout the graph in way like described in the basics. But this method requires an simulation which is not easy to compute.

4.2 Implementation

This chapter describes the implementation in detail.

4.2.1 Utilized Tools, Libraries and Technologies

This chapter gives a short overview about the technologies used and lists the hardware on which the tests and the evaluation have been executed. In Table 4.1 these are listed.

Computer Spezification	
Processor	Intel i7-5820K (12) 4.100GHz
RAM	15941MiB
Operating System	Linux 5.10.68-1-MANJARO x86_64 21.1.4 Pahvo
Graphic card	NVIDIA GeForce GTX 980
Browser	Chrome Version 94.0.4606.61 (Official build) Arch Linux (64-Bit) Firefox 92.0.1 (64-Bit)

Table 4.1: This table shows the technologies used and the hardware on which the performance will be evaluated later.

In the following a list of the libraries are presented with a short introduction:

Vue.js v2.x: Vue is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries. [vuea] It was explicitly decided to use version 2 because there is already a web application written in Vue v.2.

Bootstrap v4: With BootstrapVue you can build responsive, mobile-first, and ARIA accessible projects on the web using Vue.js and the world's most popular front-end CSS library. [vueb]

ubii-node-master: This is the server. This server is responsible for various things, such as client registration, establishing a session, assignment of devices, task scheduling of the processing modules and assignment to clients, network communication of the topics to the clients. [Weba]

ubii-web-frontend: Das ist ein bereits existierendes Webfrontend welches verschiedene Tools beinhaltet, um UBII die zu testen bzw vorzustellen.. Es wurde vor allem ein bereits exisitierende Beispiel verwendet, damit die Funktionstüchtigkeit der vorgeschlagenen Lösungbegutachtet und getestet werden kann. [Webb] *ubii-node-webbrowser*: This is an existing web frontend that contains various tools to test and present UBII. The main purpose why this libaray is used, is to use a already existing examples, so that the functionality of the proposed solution can be examined and tested. [Webc]

ubii-msg-formats: This is the repository for Ubii message formats. It contains everything about what ubii nodes and devices say to each other. It relys heavliy on Google Protobuf. Protobuf is Google's equivalent to XML. In a markup language developed by Google can structure data, which can then be translated into various programming then translated into different programming languages by the protoc compiler. can be translated. The translated classes are platform independent and offer among other things Methods for serialization. [Webd] [Buf]

docker: Docker Engine is an open source containerization technology for building and containerizing your applications. Docker Engine acts as a client-server application with: A server with a long-running daemon process dockerd. APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon. A command line interface (CLI) client docker. The CLI uses Docker APIs to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI. The daemon creates and manage Docker objects, such as images, containers, networks, and volumes. [doca]

docker-compose: Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see the list of features. [docb]

litegraph.js: A library in Javascript to create graphs in the browser similar to Unreal Blueprints. Nodes can be programmed easily and it includes an editor to construct and test the graphs. It can be integrated easily in any existing web application and graphs can be run without the need of an editor. [lit]

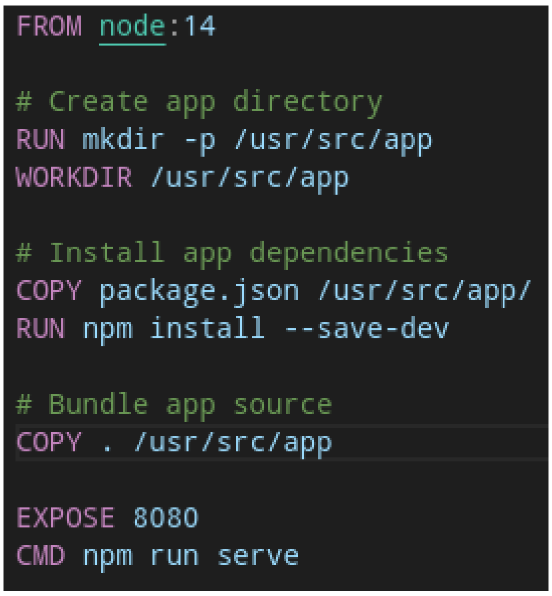
VueTreeselect v0.4: vue-treeselect is a multi-select component with nested options support for Vue.js. [tre]

d3-force: This module implements a velocity Verlet numerical integrator for simulating physical forces on particles. The simulation is simplified: it assumes a constant unit time step $\Delta t = 1$ for each step, and a constant unit mass $m = 1$ for all particles. As a result, a force F acting on a particle is equivalent to a constant acceleration a over the time interval Δt, and can be simulated simply by adding to the particle's velocity, which is then added to the particle's position. [d3f]

4.2.2 Setting up the Development Environment

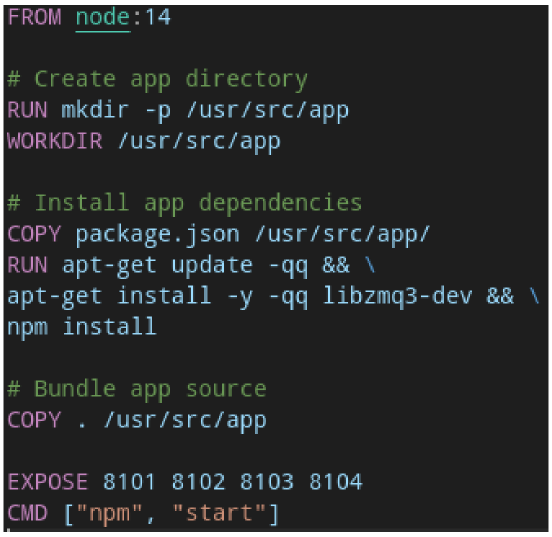
First a standalone application was written. This application offered just a graph and a test for the REST interface. It quickly became clear that this work should be integrated into the ubii-node-frontend. For this to happen, some requirements have to be met. For this purpose, a Container was created using Docker. For the ubii-node-fronted it was just necessery to install the package.json and a node instance. For ubii-node-master ZeroMQ3 has been installed on the operating system and the version of NodeJS should be greater than or equal to 14. The ubii-node-master provides an API on the following ports: 8101, 8102, 8103, 8104. After that both ubii-web and ubii-node can be started using a script written in js and executed via npm.

For the web frontend it also needs a NodeJs version greater than 14 and port 8080 should be exposed. Then with the help of $npmrunserve$ the environment can be started. To start both containers now, a docker-compose script was written which builds both docker files and expose the needed ports in virtual network. Since these two applications need to communicate with each other they must be on the same network called $ubii-network$. The current version of ubii-node is working on an updated version of ZeroMQ v.5.x. This has not been tested with the current docker scripts. Figure 4.7



```
FROM node:14

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install --save-dev

# Bundle app source
COPY . /usr/src/app

EXPOSE 8080
CMD npm run serve
```

(a) These are the steps on how the environment of ubii-web-frontend can be build. Instead of a COPY one can replace it with a git call for deployment.

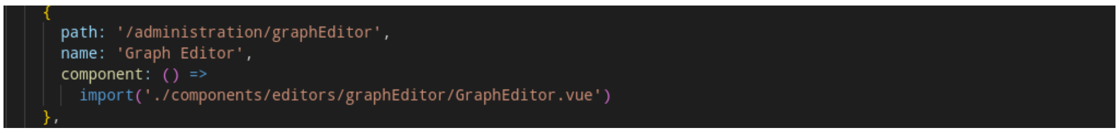


```
FROM node:14

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN apt-get update -qq && \
apt-get install -y -qq libzmq3-dev && \
npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 8101 8102 8103 8104
CMD ["npm", "start"]
```

(b) These are the steps on how the environment of ubii-node-master can be build. Instead of a COPY one can replace it with a git call for deployment.

Figure 4.7: Both pictures show the configuration files for docker.

Now the application can be made to run, but first the npm packages Litegraph.js and vue-boostrap and vue-treesselect must be installed by adding into the packag.json. To give the application a link, a route was added into the ubii-web-frontend and a new folder was created which contains the whole work that has been developed. Figure 4.8



```
{
  path: '/administration/graphEditor',
  name: 'Graph Editor',
  component: () =>
    import('./components/editors/graphEditor/GraphEditor.vue')
},
```

Figure 4.8: The route. It also shows where this work was inserted in the ubii-web-frontend.

4.2.3 Communication with the Backend and Construction Pipeline

When the page is visited, first the necessary Data structures are created. For the communication with the ubii-node-master, the UBIClientService was used which is part of the package ubii-web-browser. An instance will be created which established a connection and the connection remains as long as the applications is running and disconnect when the application is canceled. Furthermore happening in the mounting process, the object for litegraph.js is generated. This is then attached to the canvas. Litegraph.js comes with a set of already existing types. These types are not needed in this work and are therefore removed. The communication with the server, always work in the same principle. An asynchronous interface waits for a connection, if successful a service is called. Which service has been called decides the topic. The server then answer with a valid response in json format, if the request was successfully interpreted. For this work four topics were essential, namely: session, client, device as well as processing module.

After the initialization, the user is able to either create a new session or load an existing session. To increase usability, after opening the tree of the selection input mask, the program asks for all available sessions stored in the runtime of UBII and if found it leaves an entry with the name of the session in the list.

If one selects an existing session, the pipeline will be started. Roughly said, first if an session has already been loaded, it will be deleted with all its devices and processing modules. Then the graph object will be generated, after that the session will be loaded again, with it, all clients and the devices available in UBII are loaded and filtered with respect to the session name. The same will be done with processing modules. Litegraph.js needs datatypes in order to add nodes to the graph. Therefore all the Client+Devices and Processing Modules have to registered to Litegraph.js. Now it is possible to and the nodes and edges and an user is able to configure the graph. Figure 4.9

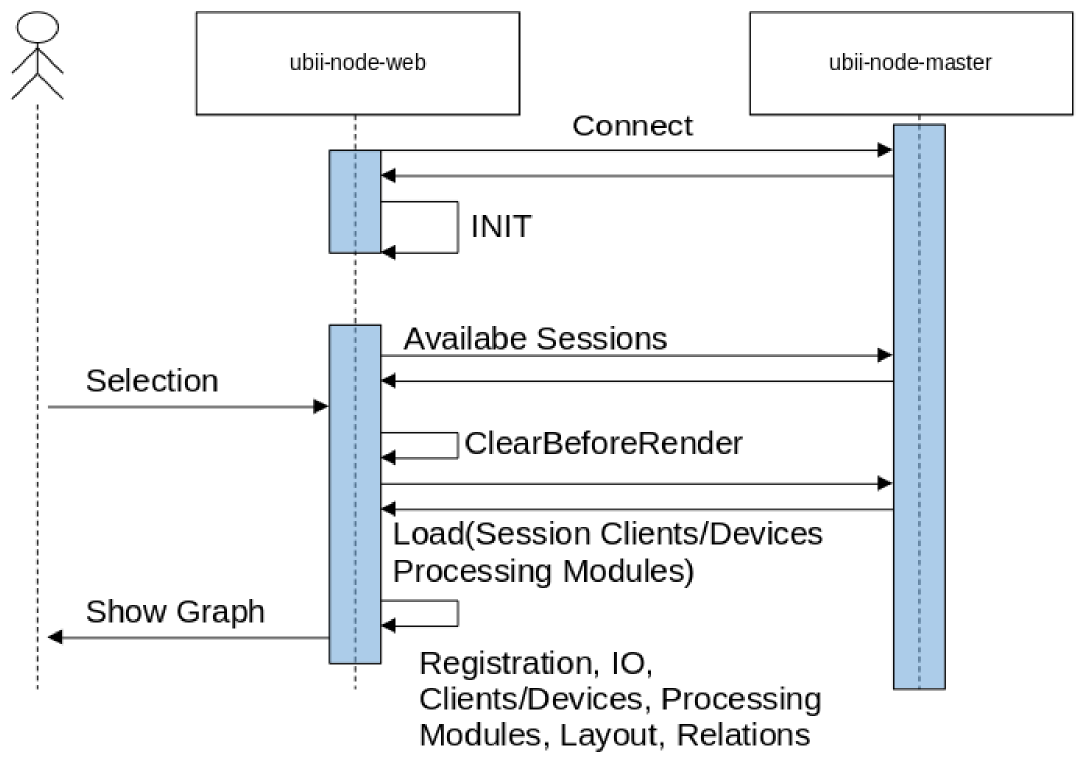


Figure 4.9: This picture shows an UML diagram. It shows the different calls made from the web browser to the server and illustrates the pipeline.

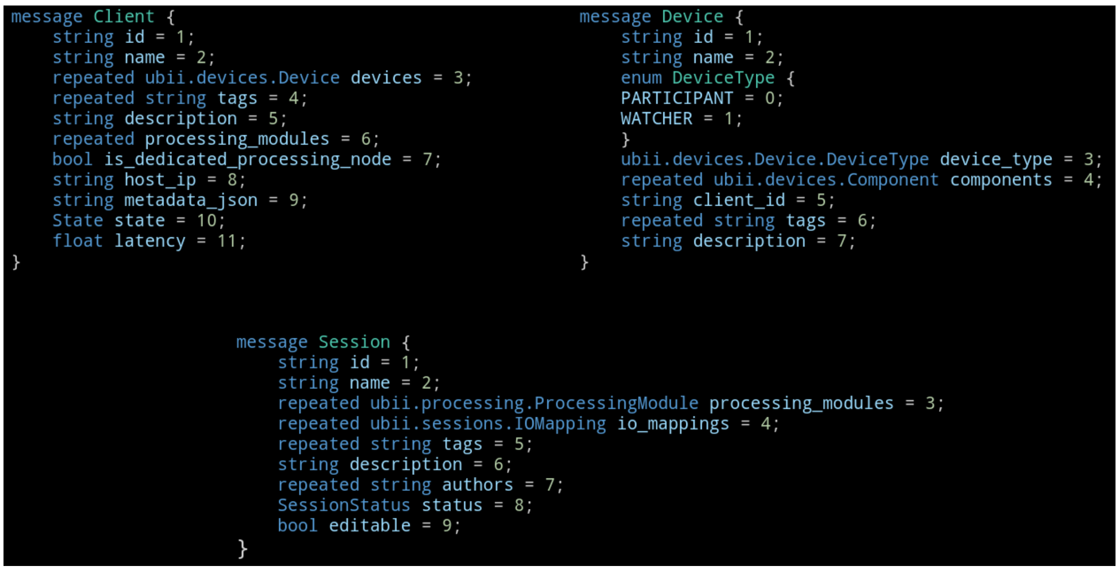
4.2.4 Parsing Clients, Devices, Processing Modules and Sessions and building the Graph

In **4.1.2** it was explained why and how the mapping UBII was chosen, this paragraph will now go into more detail about how the data structures were parsed. At first it is important to find out how data structures of Clients, Devices and Processing Modules and Sessions look like. An advantage is that the Markup Language Protobuf had been used which makes data types easy to read and understand. A Client has beside the name and ID, infinitely many devices. Besides Devices it is possible that it has also infinitely many processing modules. It also has a Boolean property that indicates whether the client is able process a processing module. In addition, a Client has an enumeration since it can be either active, inactive or unavailable.

Device also has an ID and a name, but can have two different types. On the one hand a device can be a participant, on the other hand a watcher. Participants actively participate in the UBII Network, where as a watcher only reads the Network. Furthermore it can have components. Components are generally defined by their topic and whether it is a publish or a subscribe. Besides that it also has a client id so it can be figured out which client is responsible for the device.

A processing module have both an ID and a name. Furthermore the name of authors can be given in order to find out, who wrote the function. Also it has tags and a description so the Processing Module can be categorized. Then they also have a nodeId which notes the client responsible for this object. A session id which indicates to which session the processing module belongs. A status that says if the module is currently running, initialized, stopped or destroyed. A processing mode which provides a limit on how many steps per second should be processed. Than it also have arrays for both the inputs and outputs which the processing module is subscribed to or publish. At the end functions can be written in multiple languages: C++, JAVA, Python, JavaScript and C#

Last but not least there is the session object which also has an ID and a name and also a list of processing modules. In this list a relationship is giving. This relationship gives the connection between the processing modules and the clients. In this work data muxing has not been considered because it is not quite finished yet. Figure 4.10



```
message Client {
    string id = 1;
    string name = 2;
    repeated ubii.devices.Device devices = 3;
    repeated string tags = 4;
    string description = 5;
    repeated processing_modules = 6;
    bool is_dedicated_processing_node = 7;
    string host_ip = 8;
    string metadata_json = 9;
    State state = 10;
    float latency = 11;
}
```

```
message Device {
    string id = 1;
    string name = 2;
    enum DeviceType {
    PARTICIPANT = 0;
    WATCHER = 1;
    }
    ubii.devices.Device.DeviceType device_type = 3;
    repeated ubii.devices.Component components = 4;
    string client_id = 5;
    repeated string tags = 6;
    string description = 7;
}
```

```
message Session {
    string id = 1;
    string name = 2;
    repeated ubii.processing.ProcessingModule processing_modules = 3;
    repeated ubii.sessions.IOMapping io_mappings = 4;
    repeated string tags = 5;
    string description = 6;
    repeated string authors = 7;
    SessionStatus status = 8;
    bool editable = 9;
}
```

Figure 4.10: This pictures shows the specification of Clients and Devices. As well as Sessions. The Language used in the pictures is protobuf.

Now Clients and Devices and Processing modules can be represented as nodes by following considerations.

In UBII, Clients and Devices are separated but now need to be merged to one object. In order to still have a clue which two object where merged together a new ID has been assigned. This ID is the concatenation of the two ids which a client and a device have. The name of the new merged object will still be only the name of the device since it is enough to only have the information of the device.

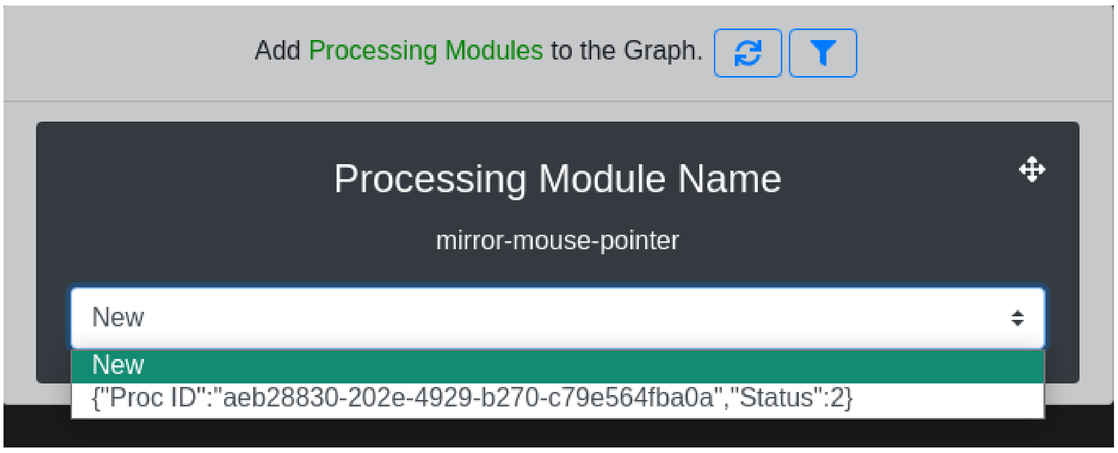
```
id: dev.clientId+'.'+dev.id,
name: dev.name
```

Listing 4.1: This Code-Snippet in /graphEditor/funcsClientProcs/clientFuncs.js is showing how an ID is choosed for the merged Objects Client and Device.

For processing modules, first a distinction has to made because either a session has been selected or a new session wants to be created. If a session has been selected, it can happen that instances of processing modules already exists, while if no session is selected only the classes i.e. the specifications are displayed. For this, a selection field has been added. With it s.o. is able to select either an already existing instance or a new instance. Processing modules have no further properties except the name and the drop down menu. Figure 4.11

```
if(session !== null ) {
uniq = [...new Map(pList.filter(val => val.sessionId === session.id).map(item =>
[item['name'], item])).values()];
}
else { uniq = [...new Map(pList.map(item => [item['name'], item])).values()]; }
```

Listing 4.2: This Code-Snippet in /graphEditor/funcsClientProcs/procFuncs.js is showing how the list for the drop down menu is produced.



{"Proc ID":"aeb28830-202e-4929-b270-c79e564fba0a","Status":2}

Figure 4.11: This picture shows the card of a processing module. In it the names has been written and a drop down menu. In this menu either an existing Processing module can selected or a new on can be created.

Now that clients, devices and processing modules have been created, they need to be registered. This means that a specification must be made so that Litegraph.js can create nodes from them. To register a Client,Device it was chosen that the title of the node should be blue. Furthermore, the constructor iterates over all components of the device to extract all input and output variables. This is done by splitting the string into several parts. Then together with the message format the connector can be constructed.

```
comp.forEach((c) => {
    if (c.ioType == 1) this.addInput(c.topic.split("/").pop(), c.messageFormat);
    else this.addOutput(c.topic.split("/").pop(), c.messageFormat);
});
```

Listing 4.3: This Code-Snippet in /graphEditor/GraphEditor.js is showing how the Inputs/Outputs of Client+Devices are being parsed.

With processing modules it is basically the same as in Client,Devices. Processing modules also have a name that is used for the title. This is displayed in green. Another difference is that a processing module has a field for the inputs and outputs. These are added to nodes using the internal name of the inputs and outputs, which is basically the variable name, the data type is extracted from the field Messageformat. Furthermore the language is extracted which can be found under the property language. This information can be accessed by double clicking on the node. Then an overlay will be opened that shows properties that cannot be changed. To be able to select the NodeId, an already existing component was used. The task of the component is to find out which clients are registered in UBII and are able to execute processing modules. To access this component, a widget had to be added to the graph, which acts as a button. This button calls a modal that executes the component. When okay is pressed, the selection is saved in the node by overwriting the NodeId. Furthermore, a field was used that indicates the number of processing steps per second. In order to be able to edit this, a widget called slider was used. With this it is possible to select a number within an interval.

```
inp.forEach((i) => {
    this.addInput(i.internalName, i.messageFormat);
});
out.forEach((o) => {
    this.addOutput(o.internalName, o.messageFormat);
});
```

Listing 4.4: This Code-Snippet in /graphEditor/GraphEditor.js is showing how the Inputs/Outputs of Processing Modules are being parsed.

Now that all client, devices and processing modules have been registered, the session can be loaded and parsed. As already mentioned, a session contains the relationships between the nodes, which can be extracted from IOMappings. But before the relationships are loaded, the client devices are filtered. This is necessary because all clients, devices registered in UBII were loaded before, if however a session is selected, only those are of interest, which actually belongs to the session. If s.o. still want to add other devices, s.o. can use the refresh button, which loads all clients and devices again. The filter process works by cutting the string again so that the ID and name is left. Then it checks if it exists in an array that contains the filtered clients and devices that are actually present in the session. These filter lists were created using the IOMappings of the session.

```
this.clientsOfInterest = cList.filter(
  (val) =>
    val.state === 0 &&
    clientFilts.includes(val.id.split(".")[0]) &&
    deviceFilts.includes(val.device.name)
);
```

Listing 4.5: This Code-Snippet in /graphEditor/GraphEditor.js is showing one of the most important data structure. It filters clients and devices which are used in the selected session

Now the nodes can be added to the graph. For the client, devices they are created using the clientsOfInterest data structure. The node in litegraph.js gets an ID equal to the composite ID. When creating the node of a processing module, the ID of the processing module is used along with the name, which are concatenated using a ".". The nodes are in an object called Graph(). In order to be able to read this information independently from the graph, an additional data structure is introduced that contains all current objects of the graph. Therefore it is important to choose the IDs of the nodes as well, because they contain the relation of the graph to the new data structure.

```
this.ClDePmNodes.push({
    edges: io,
    type: type,
    id: id,
    name: realName,
    func: func,
    procMode: procMode,
    nodeId: nodeId,
    sessionId: sessionId,
    inputs: inputs,
    outputs: outputs,
    node: node_const,
    sessionName: sessionName,
});
```

Listing 4.6: This Code-Snippet in /graphEditor/GraphEditor.js is showing one of the most important data structure. It contains the current state of the Graph in an array which can be accessed from the scope running the GraphEditor.

It is important to note that all changes in the graph must be reflected in this data structure, otherwise no reliable session can be established.

Now the nodes can be connected. For this purpose for each processing modules the inputs are taken in order to find the mappings. The same happens with the outputs.

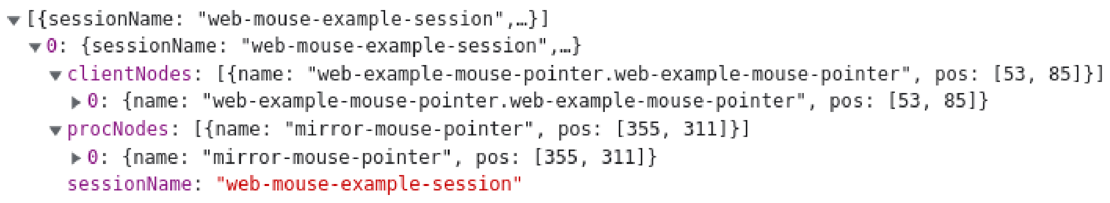
```
m.forEach((p) => {
    p.edges.inputMappings.forEach((i, index) => {
      const dc = c.filter((val) => val.edges.filter((e) => e.topic === i.topic))[0];
      if (dc !== null) dc.node.connect(index, p.node, index);
    });
    p.edges.outputMappings.forEach((o, index) => {
      const dc = c.filter((val) => val.edges.filter((e) => e.topic === o.topic))[0];
      if (dc !== null) p.node.connect(index, dc.node, index);
    });
```

Listing 4.7: This Code-Snippet in /graphEditor/GraphEditor.js is showing how the connection are readed from the processing modules.

4.2.5 Calculation of the Position of the Nodes

If a new node is added to the graph, the position of the node is $[x = 0, y = 0]$. In order to be able to determine features from the positions of the nodes in the graph, three different layouts have been implemented.

The first possibility is that the user can determine the position of the nodes himself, for this purpose the mouse can be used and the nodes can be moved and changed arbitrarily in the graph. Once all the positions of the nodes have been changed and agreed upon, the layout can be saved. It is currently not intended that UBII gets a user management, so the positions are not stored in a database governed by UBII, but in the local storage of the web browser. For this a data structure called ubi is created in which the objects can be stored. These objects differ with the session name which can then be accessed by the name. Figure 4.12



```
▾[{sessionName: "web-mouse-example-session",…}]
  ▾0: {sessionName: "web-mouse-example-session",…}
    ▾clientNodes: [{name: "web-example-mouse-pointer.web-example-mouse-pointer", pos: [53, 85]}]
      ▸0: {name: "web-example-mouse-pointer.web-example-mouse-pointer", pos: [53, 85]}
    ▾procNodes: [{name: "mirror-mouse-pointer", pos: [355, 311]}]
      ▸0: {name: "mirror-mouse-pointer", pos: [355, 311]}
     sessionName: "web-mouse-example-session"
```

Figure 4.12: The structure of storage used to save position of the nodes of the graph.

If there are too many nodes in the graph, another layout has been added. The algorithm can be used to pre-sort the nodes. The pre-sorting was implemented using litegraph.js. This sorting algorithm outputs a grid representation of all nodes. Thus this layout fulfills the criterion of orthogonal symmetry. The advantage is that this layout can be created very quickly and the graph looks orderly. The disadvantage is that in this layout no consideration is taken on the edges and thus also not possible to determine cluster formations. The structure itself remains nevertheless clear, because the edges run below the nodes, if an edge overlaps with a node. It remains to note that the main feature here are the nodes and not the edges.

For the third variant, an algorithm from the force-directed graph family was used. As described in the basics, this is a graph in which forces act on nodes. This algorithm was not self-implemented and a libarary named d3fore was used. For the input two arrays must be created. In the first array all nodes must be added. In the second array all edges are added. Nodes can also have a radius. This is important because nodes should not overlap. For this a collision detection was added. The last point is that the center is not the position $[0, 0]$ but the center of the canvas, so an offset must be added.

4.2.6 Save the Session

When all changes have been made, the session can be saved. The main data structure ClDePmNodes is used for this purpose. First it is checked whether it is a new or an already existing session. If it is an existing session, the name of the selected session is used. If it is a new one, a new session name must be chosen. Now all processing modules are extracted and the new link are readed out. Using the REST interface, the modified session is sent to UBII, which is then stored in a database for later use.

4.3 Performance Evaluation

This chapter tested the grapheditor in several ways. Like fps, time of determining the layout and layout effects.

4.3.1 Setting up the environment

Due to the lack of time no automatic test was written, but with the application was tested with a selected configuration the performance was evaluated. The procedure was as follows. First, two nodes were created and the number of frames per second of the canvas object was measured. Then different layouts were determined and measured how long the call of the ordering functions takes. The number of nodes was then increased to 9 and finally to 19.

4.3.2 Results

Number of Nodes	FPS
2 Nodes	60 fps
9 Nodes	60 fps
19 Nodes	60 fps

Table 4.2: How many fps could be achieved.

It turned out that the fps stayed constant on 60 fps. But it depends on the zoom factor. If the graph is zoomed out the level of detail will be reduced, if zoomed in the level of detail will be increased and the fps may drop. A test with a node having a lot of edges showed that the fps dropped to 30. Figure 4.13 Also another test showed that just nodes are better for the performance then less nodes but a lot of edges. Even 100 nodes could be rendered smoothly but having edges will make the graph slow. Also the animation of the flow will decrease the fps.

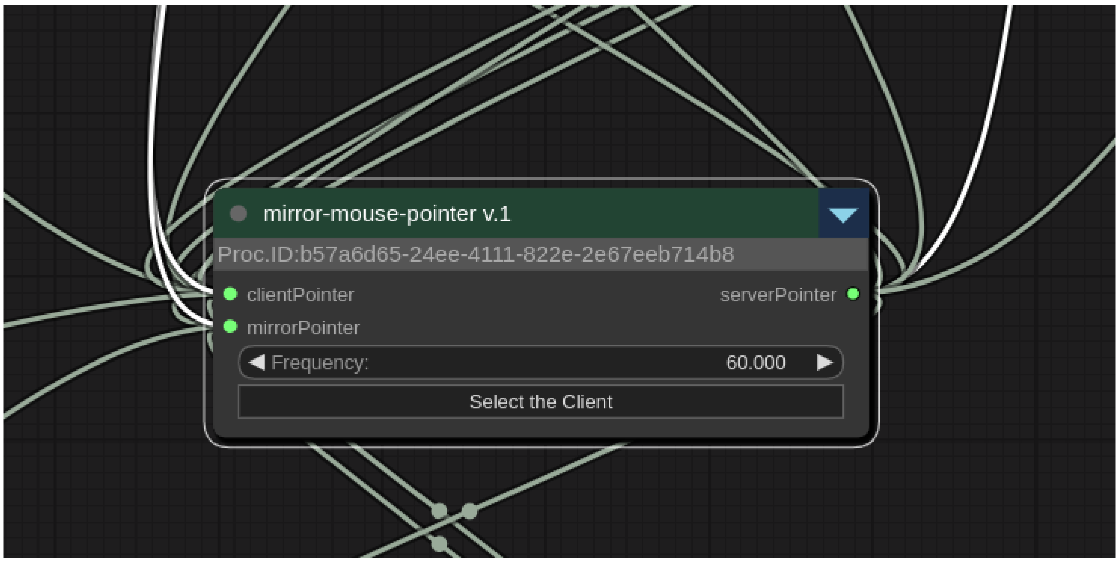


Figure 4.13: A node with a lot of edges.

The next test was how long does it take to layout the graph. It was shown that layout provided by Litegraph.js is the fastest and the force directed graph the slowest taking a constant time of around 5000 ms. The reason why the Grid layout provided by litegraph.js is the positioning of the nodes just arrange them in two or one line. The reason that loading from local storage increases for the number of nodes, can be explained by arguing that parsing JSON to objects is a hard problem and takes a lot of computing time. The same goes for calculating the position by a force graph layout, but the time can be reduced if the parameters for the simulation will be changed. Figure 4.14 The last

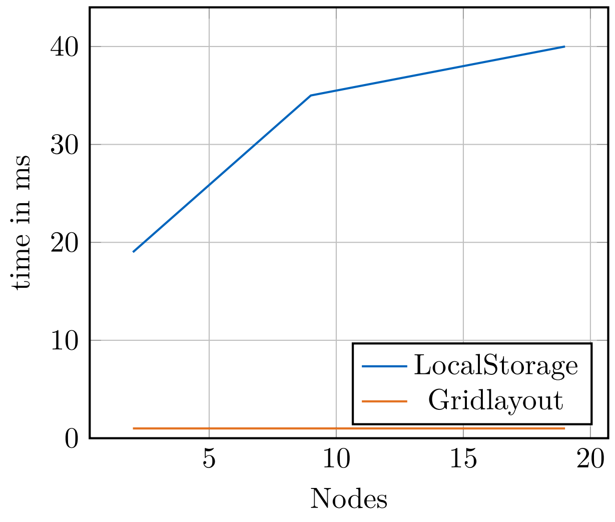
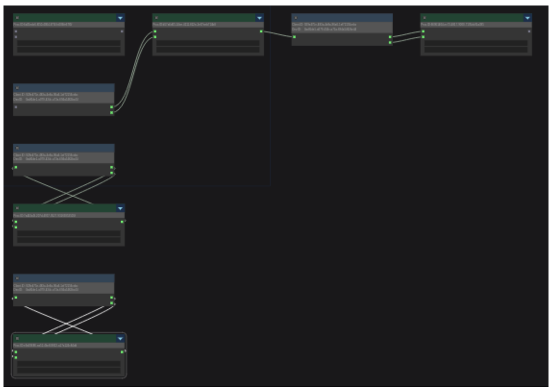
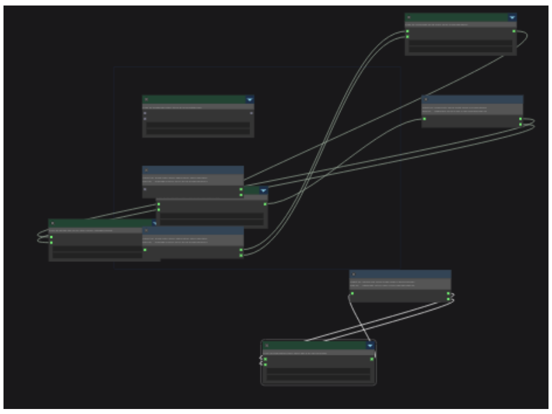
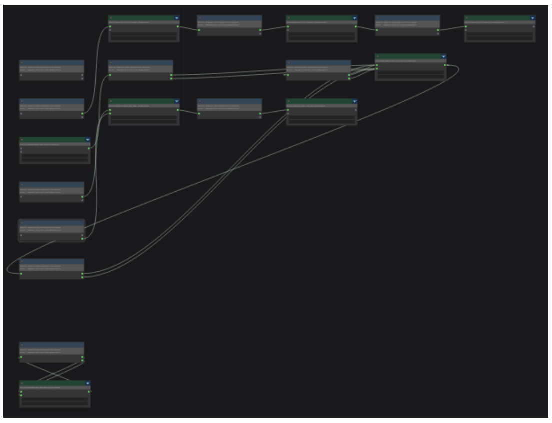


Figure 4.14: This Plot shows that loading the positions form local storage takes longer than the list layout and it increases with the number of nodes.

test analyzes the produced layout of the layout algorithms. Two algorithms had been chosen, the topological layout from litegraph.js and the force directed layout. First some qualitative measures for Figure 4.15 (a) where it is possible to see that the edges are having almost no intersection besides the cycle in left bottom corner. The nodes are positioned symmetric to the diagonal of the canvas. The distance of each node to the following node is the same for all nodes. On the other side in the layout in (b) there are two to three clusters. The nodes are overlapping a bit. The edges intersect each other. There is no symmetry. One could say that (a) can be used in order to understand the control flow and (b) can be used in order to understand the strongly connected components. For (c) and (d) the same consideration can be done but the force directed graph in this case are also revealing a control flow and isolated nodes are shown.


(a) The Layout with 9 nodes using litegraph.js


(b) The Layout with 9 nodes using a force directed graph


(c) The Layout with 19 nodes using litegraph.js

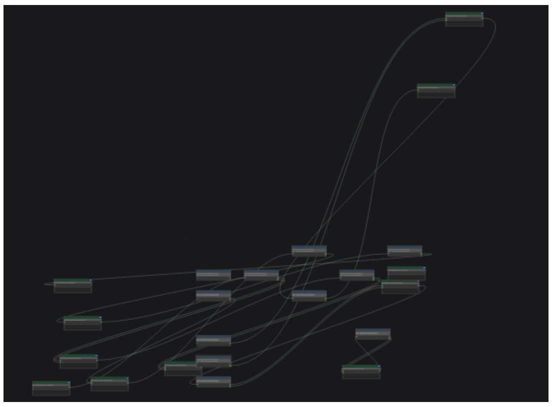

(d) The Layout with 19 nodes using a force directed graph

Figure 4.15: This figures shows 4 Graphs created with the Grapheditor. The first two graphs consists of 9 nodes and randomized connections to each other. The second two graphs consists of 19 nodes. Both Layouts, left litegraph.js internal layout and right a force directed graph can be considered as valuable since both show different features.

4.4 Tests

This chapter describes how the system was tested using an application that uses UBII as middleware. For this purpose, first the application is presented, the program will be briefly analyzed and then it is explained what criteria must be hold if the UBII graph editor is used.

4.4.1 Setting up the test environment

To test the application an already existing application was used. The application used is called Mouse Demo. The purpose of this application is to provide a tutorial. The demo consists of a client that has a device, a mouse, that sends its position data to a processing module running on a client. On the processing module the coordinates are being normalized, which then forwards the position to the client. It can also be selected if a mirroring should be done. Furthermore it is possible to specify how often per second the data should be published.

The following test criteria must be hold when using this work:

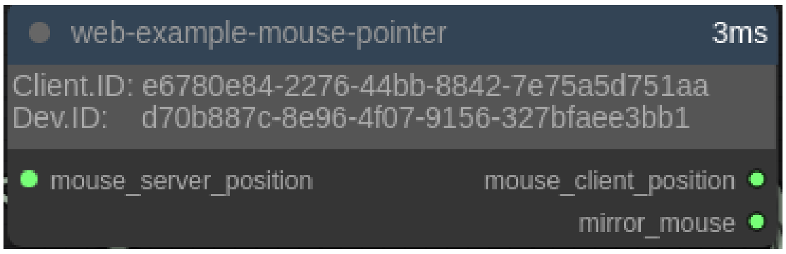
1. if the application is running it must be possible to select a session from the runtime
2. if the session is loaded for the first time the position of all nodes is $[0, 0]$
3. it can be selected how the layout should look like and the graph editor must react accordingly
4. at any time the position of the nodes can be saved, the session can be saved and a new layout can be determined at any time.
5. a client may not be connected to another client, a processing module may not be connected more than once to another processing module
6. saving a session must always be a valid session object
7. the same client, device or processing module should only exist once in the graph

5 Tools for Debugging

In this chapter it will be explained what possibilities have been implemented to detect possible errors that may occur because it is a distributed system. It is first described which functionalities have been chosen. Then the realization and problems are described in the implementation chapter. This is followed by criteria that have hold when tested.

5.1 Approach

To make it easier to debug applications or find errors, two functionalities have been added. The first function is a status logger. Since clients define an application, it must be ensured that all clients are available in a UBII network and that there is a fast connection to them. To ensure this, an asynchronous function was created for each client that periodically queries how long the client takes to respond to a PING message. In order not to query all the time, it is possible to turn the asynchronous function on and off. In the GUI a REST interface is called with the corresponding service that has been implemented for this purpose. Figure 5.1


web-example-mouse-pointer 3ms

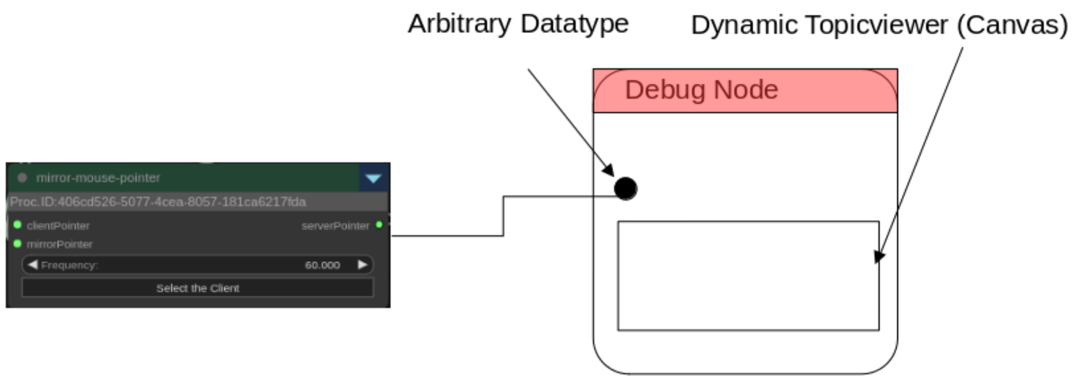
Client.ID: e6780e84-2276-44bb-8842-7e75a5d751aa
Dev.ID: d70b887c-8e96-4f07-9156-327bfaee3bb1

Figure 5.1: A Client, Devices is depicted. On the top right the latency is being printed. The user is responsible to interpret the value.

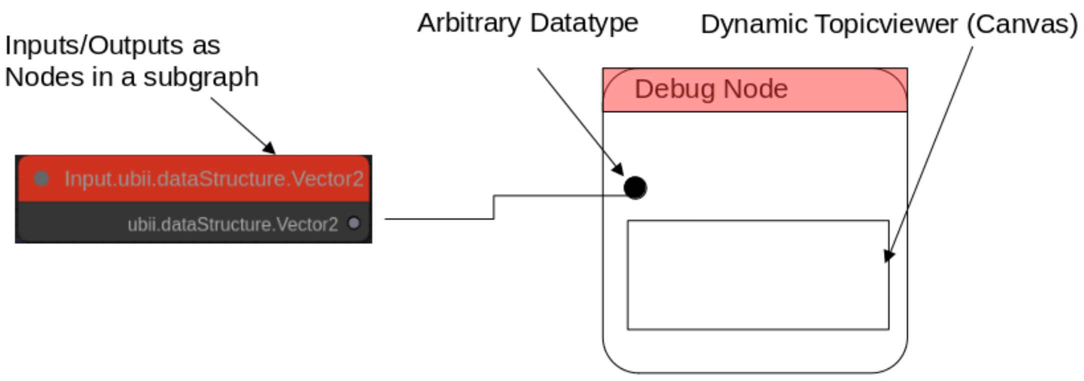
The second functionality is a message inspector. As already mentioned, processing modules can subscribe to several topics or publish topics. It would be advantageous to always be able to display the current dataset in order to detect possible errors at an early stage. An example would be a client that publishes the position of the mouse and a processing module that processes this data, but mirrored although the flag was not set. With the help of the Message Inspector the error could be detected early.

To make this possible, several variants were experimented with. The first variant provided that for each graph of a session, also one or more debug nodes exist, which can be used to visualize topics. These have only one connector as input, which however had a dynamic type. This means that if an input was a Boolean, this was recognized and the node reacted accordingly. A client or a processing module could cut its connection in the session and connect to the debug node which then visualized the data if the play button was pressed. After the final implementation it was found that it is unnatural to misuse an editor designed to configure a session in such a way that it also serves as a visualizer for topics. Figure 5.2

The second variant was relatively similar to the one just presented, except that instead of using the graph that should be used to configure a session, a new graph is built that opens in a new window when a button is activated. A so called subgraph. In this subgraph there were no nodes of the type Client, Devices or Processing Modules anymore, but the data types used in UBII as nodes which could be generated from the inputs and outputs of a processing module and again the Debug Node that was discussed before. After the implementation was finished, it was found that it was relatively tedious to manage the subgraphs. This was implemented in way that for each node in the graph exist new subgraph, which in turn contained new node types. Figure 5.2



(a) The first variant of the message inspector. Processing modules or client, devices could connect to debug nodes in order to visualize the data.


Inputs/Outputs as
Nodes in a subgraph
Arbitrary Datatype
Dynamic Topicviewer (Canvas)
Debug Node
Input.ubii.dataStructure.Vector2
ubii.dataStructure.Vector2

(b) The second variant. After clicking the button of a processing module, a subgraph will be opened, contain UBII data structures as nodes, mapped from the in and outputs of the processing module.

Figure 5.2: The two Variants of the message inspector.

For the third variant, which was also used in this work, a button was added to each processing module, which can be activated. If the play button is active and the play button is pressed, all incoming and outgoing, connected data flows are visualized. Figure 5.3

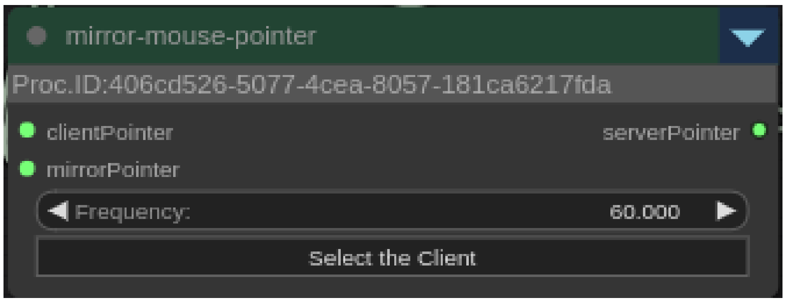


Figure 5.3: A processing module with a button on the top right.

For the Message Inspector to work, a visualizer must be implemented for each data type that exists in UBII. Exemplary it was done for the data types Boolean and Vector2D. For the rest a template was created. The Message Inspector works in such a way that if an edge exists between two nodes, the Message Inspector can be called via a processing module, which then displays the visualizer for all in and outputs. These are cards that contain a canvas. For a Vector2D the coordinates are normalized and the position is displayed with a green dot. For a Boolean a string with the value false or true is printed. For better identification the colors green and red are used. Figure 5.4

For the other data types no inspector has been implemented yet. It is also thought that these cards can be extended arbitrarily, e.g. instead of normalized coordinates, a maximum and a minimum can be specified, which interpolates the coordinates accordingly. If a processing module has a lot of inputs and outputs, the edge can be clipped in the graph, which removes the corresponding visualizer in the message inspector.

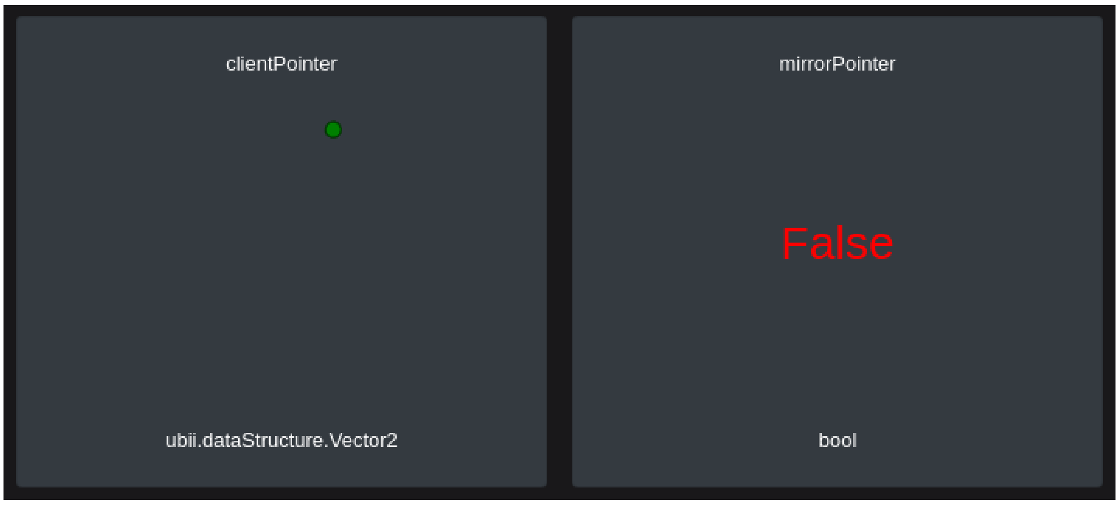


Figure 5.4: This pictures shows the cards used as message inspectors. In the left a green dot represents coordinates. In the right a boolean is printed.

5.2 Implementation

This chapter deals with the implementation of the status logger. It is described what had been changed in UBII, so that the latency of a client can be checked and/or the stability. Then it is shown how the Message inspector has been implemented.

5.2.1 Network latency

To find out the latency between a server and a client, a new class was created in ubi-node-master. This class contains a map in which all active requests are stored. In order to make the client manager aware of this, an additional call to the status logger is made in the life monitoring process of each client. The implementation provides that if a client is actively participating in the network, it automatically responds to a PING sent by the server at regular intervals. If UBII receives a PONG message from a client, the function addLatency can be called in the new class. This function asks if the map already contains an active PING and if so, it updates the client's latency value with the difference between the timestamp stored in the map and the current timestamp. As mentioned before, an asynchronous process runs in the GUI that constantly asks for the updated latency in a ten second interval. To make this possible, an additional service had to be written, whose task is to return the latency of all clients. Instead of returning all information, only the ID of the client and the latency is needed. In the frontend it works like this, that all latency updates are stored in a list, for the correct output of each client the corresponding ID is searched in this list in order to find the corresponding latency.

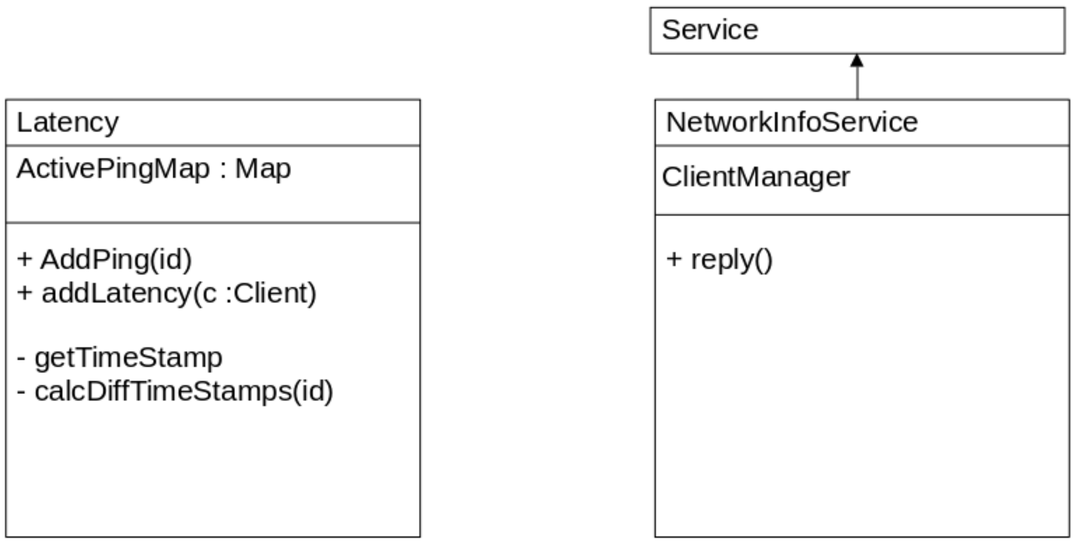


Figure 5.5: This uml diagram shows how the lantency logger was implemented in ubii-node-master.

In order to call the new service, a new REST interface was built by changing the ubii-message-formats. This means that a new service has been created under constants.js, which can then be built using the proto compiler (protoc) and a build script.

```
"LATENCY_CLIENTS_LIST": "/services/latency_clients_list"
```

Listing 5.1: This Code-Snippet in messageformats/constant.js is showing REST interface for the latency.

5.2.2 Message Inspector

To implement the message inspector several variants were tried. The first variant was to use a Vue.js watcher. This means that there is a data structure on which an observer is registered and if the data structure changes or updates with new values, an event is triggered which can be implemented by a callback. More precisely, each processing module has as many data structures as in and outputs and if the observer detects an update of a data structure the callback is executed. The problem with this type of implementation is that a copy operation has to take place which takes an extremely long time. This process took so much time that it caused dropouts in the visualizer, so this variant was discarded.

Therefore, it was decided to use a different method. First, an abstract class or component called Topicviewer was created. If s.o. activate the message inspector in the GUI, a new instance of the topicviewer is created for each topic of the processing module. In order to determine the type and the corresponding name of the topic, the attribute of the connectors are used. If the instance is created successfully, the corresponding card is added to the row, specified with HTML, using appendchild. However, this also means that HTML and Javascript is created at runtime, which is a security vulnerability, if UBII sends data that contains malicious code, the data has to be escaped. To identify these instances, they are assigned an ID that Vue.js automatically creates with each new component. When mounting the topic viewer it checks what type it is and depending on that different visualizers can be used. As already mentioned, for the type Vector2D, for example, a canvas has been chosen, which marks the coordinates with a green dot. For the data type Matrix a canvas could be used, if the matrix should be interpreted as an image. UBII offers, after a subscribe or publish callbacks that should be called on incoming or outgoing data streams. This was used for the TopicViewer, in order to provide a direct call of the function without intermediate steps, which improved the performance significantly. Figure 5.6

```
this.debug.active_inputs.forEach((val) => {
    UbiiClientService.instance.unsubscribeTopic(val.topic);
  });
  this.debug.active_inputs.forEach((val) => {
    UbiiClientService.instance.subscribeTopic(val.topic, (data) => {
      switch (val.type) {
        case "ubii.dataStructure.Vector2":
          val.component.ubii_updateVector2(data);
          break;
        case "bool":
          val.component.ubii_updateBool(data);
          break;
      }
    });
  });
```

Listing 5.2: This Code-Snippet in /graphEditor/GraphEditor.js is showing how the message inspector is filled with data coming from UBII.

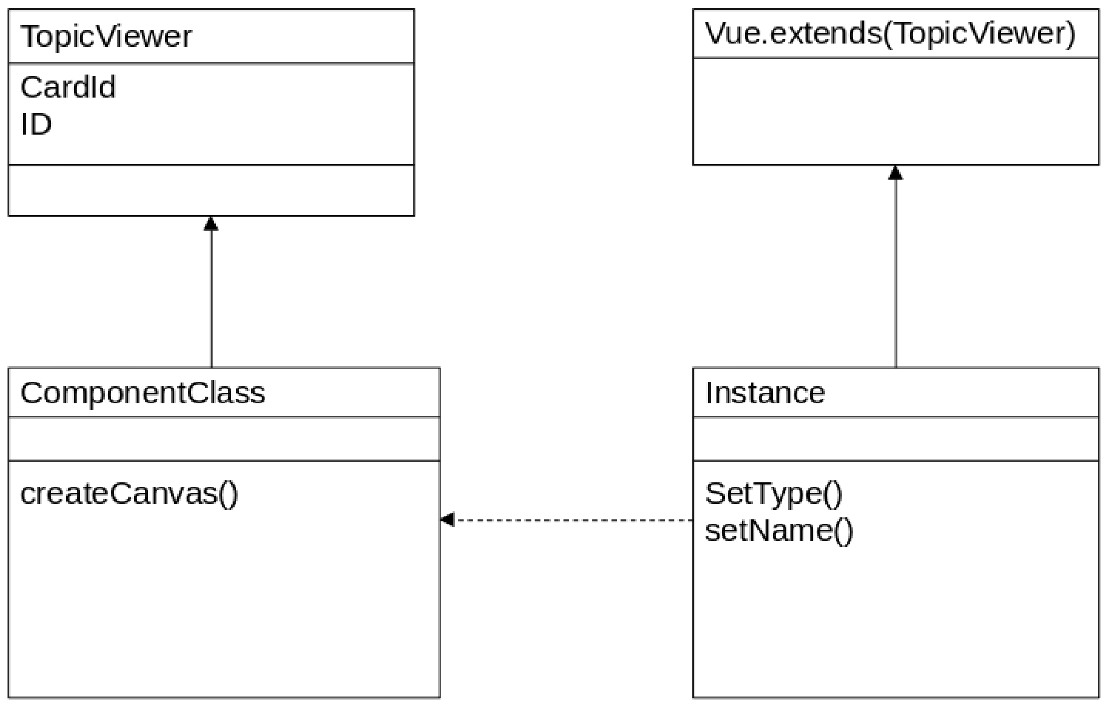


Figure 5.6: This uml diagram shows how the factory pattern was utilized to implement the message inspector in the third variant.

5.3 Tests

This chapter describes how the system was tested using the same application as in **Chapter 4** which is why the program is not explained anymore. But also here criteria are mentioned which must hold.

5.3.1 Setting up the environment

As already mentioned, the mouse example was used again for this test. The mouse example needs a client and a device, which is the mouse itself, to function properly. Pressing the play button displays the connection quality on the right side of the node. Both UBII and the web application run on the same client and in different docker containers respectively. Therefore, it can be explained why the latency is relatively low at 1 ms. Now it is possible to check the messages that are sent. Firstly, it turns out that if the play button was activated, changes are not immediately applied. This means that if e.g. the boolean flag mirror is not used in the session, it is not possible to find out which status it has. This is because the callback which is used, because only publishes changes and not initial values.

The following criteria must hold:

1. if a link is created or removed the corresponding topicviewer will be created or removed too
2. if another processing module than the currently active one is clicked, the scene of the message inspector has to be changed.
3. the client may only update the latency if the play button is active
4. if a new client, devices or processing modules are added to the graph the message inspector must be paused and can't be called anymore because the session must be saved first.

6 Summary and Conclusion

This chapter gives a conclusion. Future work that may be done is then discussed.

6.0.1 Summary and Conclusion

This thesis makes a proposal on how to visualize a distributed system using a graph. The basis for a distributed system is UBI. In the basics it was discussed what UBI is and what possibilities there are to arrange graphs in a way that they increase the understanding of a distributed system. Furthermore it was discussed what the difference is between debugging and debugging distributed applications. The related work provided insight into how a graph layout can be analyzed. Furthermore, they gave insight into the importance of implementing logging system. Finally, a mapping from UBII to a graph was shown and then different user interfaces with their interaction possibilities were presented. It was explained in detail how the application was implemented and what alternatives there were in the implementation. Finally the performance was evaluated and it was shown that the application runs relatively smooth if only nodes are used and not too many edges, it was shown why and how long it takes to create layouts and it was analyzed how well a layout can contribute to understanding. To detect errors early two suggestions were made. First, the latency of each client available in the network should be displayed to draw conclusions about the accessibility. Furthermore, it is important to keep track of the data sent over the network, which is why a message inspector was implemented.

6.0.2 Future Work

There is still lot of improvements that can be done. Also a lot of new features can be added. Some of the most important are now being listed:

Implementation of all visualizers for the message inspector and performance evaluation of the topicviewer: Due to the lack of time only two Datatypes have a representative message inspector. This should be done for all Datatypes supported by UBII. Some are easy to implement and visualized like for example an integers. Others are more challenging like 3D Objects. Especially if advanced rendering methods are used, the performance of the topicviewer needs to be improved.

Integate it more with UBII instead of just visualizing: Some Services were not integrated with the Grapheditor. Like stopping and starting a session are not being used. Also updateing a runnig session is at the moment also not possible.

Refactoring the implementation: It was written that a lot variants existed, all of the variants had been implemented but only one variant made it. Such processes always makes code a little bit unstructured and therefore should be refactored.

Updating the docker images to zeromq5-dev: At the end of these work, the main branch was merged into the develop branch and a lot of improvement of UBII had been done, such as for example the upgrade of libraries which made the docker image not working anymore due to a missing linked library.

Implementing more layout strategies: At the end almost all possible layouts should be tried out in order to find best suited.

Implementing more debugging strategies: In the related work it is noted that algorithm exist which can possibly or definitely say that some state of variables where at some point in time true. Such a monitor system would be useful for debugging and could be added. This would mean all messages should also been send to a centralized server which task it is to monitor the distributed system.

List of Figures

List of Tables

Bibliography

[11] *The WebSocket Protocol.* `https://datatracker.ietf.org/doc/html/rfc6455#section-1.1`. [Online; accessed 13-October-2021]. December 2011.

[Agi02] D. J. Agins. *Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems.* 2002.

[Bin+19] C. Binucci, U. Brandes, T. Dwyer, M. Gronemann, R. von Hanxleden, M. van Kreveld, P. Mutzel, M. Schaefer, F. Schreiber, and B. Speckmann. "10 Reasons to Get Interested in Graph Drawing." In: *Computing and Software Science: State of the Art and Perspectives.* Ed. by B. Steffen and G. Woeginger. Cham: Springer International Publishing, 2019, pp. 85–104. ISBN: 978-3-319-91908-9. DOI: `10.1007/978-3-319-91908-9_6`.

[Buf] P. Buffers. *Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.* `https://developers.google.com/protocol-buffers`. [Online; accessed 13-October-2021].

[CBB] B. Cheswick, H. Burch, and S. Branigan. *Mapping and Visualizing the Internet.*

[Cou+12] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *DISTRIBUTED SYSTEMS Concepts and Design.* Addison-Wesley Professional, 2012.

[d3f] d3force. `https://github.com/d3/d3-force`. [Online; accessed 13-October-2021].

[doca] docker. `https://docs.docker.com/get-started/overview/`. [Online; accessed 13-October-2021].

[docb] docker-compose. `https://docs.docker.com/compose/`. [Online; accessed 13-October-2021].

[dra] drawflow. `https://github.com/jerosoler/Drawflow`. [Online; accessed 13-October-2021].

[HHE06] W. Huang, S.-H. Hong, and P. Eades. *Layout Effects on Sociogram Perception.* 2006.

[Igu] S. Iguana. *Large Graph Visualization Tools and Approaches*. https://towardsdatascience.com/large-graph-visualization-tools-and-approaches-2b8758a1cd59. [Online; accessed 13-October-2021].

[lit] litegraph. https://github.com/jagenjo/litegraph.js?files=1. [Online; accessed 13-October-2021].

[Mil10] I. Millington. *Game Physics Engine Development*. 2010.

[PRM95] H. Purchase, R.F.Cohen, and M.I.James. *An Experimental Study of the Basis for Graph Drawing Algorithms*. 1995.

[ret] rete. https://github.com/retejs/rete. [Online; accessed 13-October-2021].

[S+08] J. A. S., F. Karray, M. Alemzadeh, and M. N. Arab. *Human Computer Interaction: Overview on State of the Art*. University of Waterloo. 2008.

[Sec] S. Sechrest. *An Introductory 4.4BSD Interprocess Communication Tutorial*. https://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf. [Online; accessed 13-October-2021].

[tre] treeselect. https://github.com/riophae/vue-treeselect. [Online; accessed 13-October-2021].

[vuea] vue. https://vuejs.org/v2/guide/. [Online; accessed 13-October-2021].

[vueb] vue-bootstrap. https://bootstrap-vue.org/docs. [Online; accessed 13-October-2021].

[Weba] S. Weber. https://github.com/SandroWeber/ubii-node-master. [Online; accessed 13-October-2021].

[Webb] S. Weber. https://github.com/SandroWeber/ubii-web-frontend. [Online; accessed 13-October-2021].

[Webc] S. Weber. https://github.com/SandroWeber/ubii-node-webbrowser. [Online; accessed 13-October-2021].

[Webd] S. Weber. https://github.com/SandroWeber/ubii-msg-formats. [Online; accessed 13-October-2021].