Bachelor Thesis

on the topic

"Invention of a working prototype to demonstrate the IOTA Streams and Wallet protocols for a meter with additional focus on economic efficiency and the technical preparation of scalability."

by Friedrich Rosenkranz
Westsächsische Hochschule Zwickau
bioxX systems GmbH

SELBSTSTÄNDIGKEITSERKLÄRUNG GEM. § 14 ABSATZ 5 BPO

Hiermit versichere ich, Friedrich Rosenkranz, dass ich die vorliegende Bachelorarbeit mit dem Titel

“Invention of a working prototype to demonstrate the IOTA Streams and Wallet protocols for a meter with additional focus on economic efficiency and the technical preparation of scalability.”

selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

....

Ort, Datum Unterschrift

MOTIVATION

This work is part of a larger project to develop a real-time distant-reading water meter with sophisticated cryptographic security standard. It serves as groundwork for in-depth technical and economical considerations. It builds on an internship, and broadens and deepens the knowledge gained, funnelling it into this work.
The project makes use of a fair portion of the IOTA framework, a cryptography framework, including the Client, Streams, Stronghold, and Wallet protocol.

The scope of this work includes refining skills in Rust, gaining deeper knowledge and practice in programming with the IOTA Streams library, establish basic experience with the IOTA Wallet and the Tokio library, the hardware interfaces I2C and MBus, as well as shining a light into the economical aspects of the topic, outline a business model, designing a product, understanding the market, and getting an overview over financial planning and financing.
A good half of the time has been dedicated to programming, and remaining time was spent equally in research of the given topics and outlining a business model, with help of different mentors.

ACKNOWLEDGEMENT

I would like to acknowledge and give my sincere thanks to all who have helped and participated in making this work as interesting and comprehensive as it has grown to be.
Firstly, I would like to extend my thanks to my mentor Karsten Butze, who has encouraged me to pursue this project beyond an ordinary work and provided me with both materialistic support and fruitful discussions. These dialogues have widened my view many times and I cannot stress enough how much I've grown as an individual because of that.
My thanks go to Prof. Dr. rer. pol. habil. Dr.-Ing. Tobias Teich for giving this topic a platform in the form of mentoring this work and providing opportunities to get early, valuable feedback on the larger project behind this work.
I thank Dipl. Ing. Christian Flegel and Cristian Butze for continuous support as sources of feedback, technical insights, and especially Mr Flegel for designing and assembling a PIC-circuit prototype, making the technical realisation possible.
I would also like to thank all the other supporters of the project and this work through various means.
Dipl-Ing. Jürgen Schleier, Markus Stieber, and Robin Seifert from the Wasserwerke Zwickau GmbH for giving valuable feedback on an early iteration of the prototype and especially Mr. Seifert for taking the time to have an interview, that gave great insight into the current regional circumstances.
Ms. Christina Militzer, Network Representative for the SAXEED founder network and Chantal Kling, for long evaluation meetings and guidance through the process of applying to a scholarship.
Mr. Jeroen van den Hout from the IOTA Foundation, who always had an answer to my questions regarding the IOTA libraries.

ABSTRACT

In this project we develop an intelligent water meter based on software solutions offered by the IOTA Foundation. The water meter allows the customer to map water usage in real-time and pay water on demand, as well as the water provider to map water usage on a greater scale, regulate water supply during low-demand phases and offers regulatory functions to prepare for drought or humid climate, and to incentivise sustainable water usage in high-demand fields like agriculture. This functionality is phrased into a research issue: **Invention of a working prototype to demonstrate the IOTA Streams and Wallet protocols for a meter with additional focus on economic efficiency and the technical preparation of scalability**.

Utilizing the IOTA streams protocol, a next generation secure data connection is established between the water meter and a server-sided software application. On this connection, water consumption is mapped into a data bench, and informative data and commands are issued to the graphical interface of the meter.

The IOTA wallet library is leveraged to provide customer accounts corresponding to their meter. IOTA tokens can be send to the account, which grants access to water in a matter of seconds. Depending on the regulatory scenario, water flow can be stopped as soon as the account is exhausted (i.e. public well), or an overdraw can be established in order to guarantee fulfilment of basic human rights (i.e. private households).
Since pricing data can be calculated server-sided and water consumption is mapped in very narrow intervals of as low as 4 seconds, the price can be used as tool to regulate consumption.

The physical components include an electrical ball valve to shut down water flow automatically, a command line interface to provide informative data,, a Raspberry Pi running the client-sided software application, and a water meter with MBus-Interface, as well as a Controlling Board to connect the Raspberry Pi with both peripheral devices.
The finished prototype shows, that water consumption can be mapped on a highly secure level, in near real-time, from afar, flexible for most applications.

STRUCTURE

1. Theory

Chapter 1 presents a scientific base of knowledge and introduction into the topic of blockchain technology, the IOTA protocol, all leveraged hard- and software interfaces, as well as a short overview about the underlying topic of water consumption.

1.1. Blockchain Technology

Over the past years, more and more media attention has been drawn to cryptocurrencies (CCs) like Bitcoin (BTC) or Ethereum (ETH). Primarily seen as decentralized financial medium, they posses an intricate characteristic that makes them an interesting subject for communication and data transfer.
This characteristic is the encryption algorithm and validation principle, that makes CCs decentralized and is often referred to as Distributed Ledger Technology (DLT). It first emerged in 1991 and 1996 by Haber, Stornetta, and Anderson[1] and proposed the concept of the distribution of common ledger entries, transactions, among all network participants.
2008, the Bitcoin whitepaper was released[2]. This marks the begin of the Cryptocurrency era with a Proof-of-Work (PoW) concept proposed by Satoshi Nakamoto, in which a transaction was encrypted in a way that it contained hashed information about its predecessor. By comparison, a network participant can verify the legitimacy of the transaction. The name blockchain also derives from this trait, that all transactions are virtually chained to each others by hashes, grouped in blocks of thousands of transactions.

1.2. IOTA

The IOTA framework is a DLT protocol published in 2015. It is supervised by the IOTA foundation and offers several protocols with different goals[1]. It's not a blockchain, but a Directed Acyclic Graph (DAG), which poses an interestingly important difference to classical PoW-blockchains in regard to transfer speed and capacity.

Figure 1[3]: Examples of IOTA use-case targets. Smart and IoT-related sensor networks stand in the focus of high-security data transmission, and are main focus in this work, too.

In this work, we rely heavily on the Streams protocol and incorporate the Wallet protocol for further functionality.

CHRYSALIS

When we refer to Chrysalis, we mean a series of updates to the IOTA Mainnet, resulting in an upgrade to the IOTA 1.5 Mainnet. The Chrysalis updates have been working on a range of topics regarding the scalability and being enterprise-ready[4]. As the IOTA network operates with an entity called Coordinator it is not decentralised by definition. The Coordinator frequently validates a transaction which serves as anchor point for following transactions. Disconnecting the Coordinator from the Mainnet makes the network fully decentralized and is goal of the Coordicide update, as explained later. In order to be able to transition to the IOTA 2.0 Mainnet, several roadmap stops have been defined as required by the IOTA Foundation before a transition can be undertaken. Here's an overview over the necessary steps to make the Mainnet ready to be independant of the Coordinator[5]:

- White-Flag Approach
- New Milestone Selection Algorithm
- Uniform Random Tip Selection (URTS)
- Replace Winternitz One Time Signature Scheme with Ed25519 Signature Scheme
- Use simpler Atomic Transactions (ATX)
- Switch to Unspent Transaction Output (UTXO) account model
- Switch to Binary Representation (from ternary)
- Autopeering
- Improved documentation and API

Figure II[6]: The steps towards Chrysalis depicted in an infographic

For more details about the different steps, we recommend reading into the different research papers at iota.org/foundation/research-papers or the IOTA wiki at wiki.iota.org. Many of the defined requirements have already be implemented[6], for example URTS, White Flag, and milestone selection in the first phase, and ATX, Ed25519 Signatures, binary representation, and updated APIs and dev tools with the second phase. The Chryssalis update also paves way for several cryptocurrency-related functionality, like smart contracts, non-fungible tokens (NFTs), and tokenization, which run on the Shimmernet, also implementing the Shimmer currency. The Devnet for Coordicide has already launched and is currently used to re-evaluate changes and flesh out a Mainnet release[7].

CONSENSUS PROTOCOL

The IOTA network bases their data security on a node-network, ran by network participants called the Tangle. The foundation constantly aims at improving their consensus protocols and has come up with these major principals. They vastly differ from classical approaches like Proof of Work (PoW) and newer approaches like Proof of Stake (PoS), used in blockchains like BTC and ETH[1].

As general concept, the tangle consists of network nodes that are able to validate a transaction by has-comparison. Once a message enters the tangle, it must validate two pending messages, whereafter its status switches to pending itself, making it open to validation. Validating can be seen in comparison to hashing in PoW-based blockchains, but instead of queueing thousands of transactions until a block is filled, the DAG is able to process a number of messages equal to the amount of newly entered message, or number of network nodes, whichever is higher, due to each message validating two other messages and two validations being needed to get validated. Processing a message on the tangle takes about 10 seconds, according to the IOTA Foundation[4], which holds up to practice. Since no heavy computing power is necessary without a PoW-based validation process, the DAG offers high transmission speed with potentially high message density, which is dependant on the network size,with more nodes leading to a higher density and somewhat higher speed.

Since more than one validation about a message is done, no one source of truth exists as in other blockchain networks. A message may be validated by one peer and then rejected by another. This is where IOTA implements two main concepts to form a unified opinion about the state of a message.

As main function, all network nodes are able to propagate gossip to each other. Gossip is the opinion about a transaction, which can be changed and is voted with. This basic behaviour enables the main consensus protocol in the IOTA network.

FAST PROBABILISTIC CONSENSUS PROTOCOL (FPC)

As long as no conflict about the state of a transaction occurs, no consensus mechanism is necessary. At the time a conflict is detected, the Fast Probabilistic Consensus Protocol (FPC) runs. Every node is tasked to formulate an initial opinion about the transaction in question. This opinion takes into account the timestamp and/or the account balance. Once all nodes had time to formulate an opinions, every node queries a number of random different nodes about their opinion. It adopts the median opinion as its own. This process is

repeated for each node, several rounds until the opinion of a node hasn't changed for a number of rounds or a maximum of rounds have elapsed. The threshold for how many votes a node needs to adopt an opinion changes randomly, which implements high resistance against byzantine attacks[8, 9, 10]

In order to create additional resilience against misuse, the IOTA 2.0 implementation of the FPC will also make use of Mana[11]. Split into two categories, this resource aims to reward well-behaving nodes, i.e. those nodes whose opinions end up being the end result for a transaction across the network. Nodes with higher Mana enjoy higher trust in the network, represented by being chosen more often for validation, opinion query and voting mechanisms[12].

1.3 Leveraged Interfaces

In this Chapter, we try to give a basic understanding of the different underlying protocols used in this work. For hardware protocols, a technical focus is given, while we try to also shine a light onto the conceptual principals of the IOTA libraries.

MBus

The MBus protocol, developed by Horst Ziegler with Texas Instruments[13] in 1992, is conceptualised as master-slave protocol operating on a dual-wire Serial Connections[14], much like the later explained I2C-protocol.
It supplies up to 250 slave devices and focusses on connection reliability and data security, rather than transmission speed[43]. For example, it includes a checksum and the Telegram includes more fields than a I2C-data set, which enable checks performable by both master and slave to validate data. These checks are namely parity checks, check sum checks, and length checks[14, Ch.4, pg. 5.4].

MBus vs ModBus

The main difference lies in their structure: Both are called bus protocols, though the ModBus protocol natively isn't. It is normally used as ModBus RTU protocol, which involves connecting the Bus to a RS485 (UArt) plug, which accepts up to 63 connections, effectively making the ModBus RTU a Bus structure. ModBus is the lesser used protocol and supports lesser peripherals, which is why we opt for MBus here, which also offers wireless communication with norm EN13757-4. This could be vital in upgrading large networks.

I2C

As Master-Slave-Bus, the I2C-protocol represents one of the most often used low-level electrical interfaces, mainly for intra-board communication. It consists “of a clock bus wire and a data bus wire which both form a wired logic function”[15, p. 3], as described in the patent assigned to Philips Corp. in 1987. I2C offers up to 3.4 MB/s data transfers bidirectionally and up to 5 MB unidirectionally, while supporting up to 1,024 targets [16, p. 7, 15].

Aa mentioned, a serial cock wire (SCL) and a serial data wire (SDA) connect the I2C-master (also: controller) and the I2C-slave (also: target) to a current source or pull-up resistors to ensure a HIGH, when the bus is free[16, p. 8]. A HIGH in this case isn’t a fixed current, but rather dependant on the supply voltage. A LOW is represented by 0.3 V_{DD} with V_{DD} being the device-dependant supply voltage. 0.7 V_{DD} represents a HIGH.
That makes the I2C-interface flexible in terms of connectable technology devices, that have different voltage output.

I2C follows a standardized Communication sequence. Beginning with a Start Bit (S/Sr) and ending with a Stop Bit (P), the protocols expects the slave device to send Acknowledge Bits (ACK) after each byte, or to forca a halt on the transmission by a LOW on SCL by the slave device. The bus line stays busy (Bus LOW) during that time. If the designed condition for an ACK, being a stable LOW during the ninth SCL Bit’s HIGH period, respecting set-up and hold times, isn’t fulfilled, the bit is read as a Not Acknowledge Bit (NACK). A controller can then either abort, retry or move onto the next message and potentially free the bus[16, p. 10ff.].
Since a bus layout extends the data stream to all physically connected slave devices, and the possibility of different slave devices, it is necessary to address the correct device. For that, a 7-bit sequence following S gives room for 128 (2^7) different addresses. For the rare case of requiring more address space, 8 bits following the first ACK can be leveraged to extend the address space to 1,024 (2^{10}).
Following the 7-bit address, the eighth bit indicates the direction of the data, i.e. wether the master intends to write (ONE) or read (ZERO). The ninth bit is the first ACK, after which the desired amount of bytes, seperated by ACKs is sent.
A transmission is always ended with a P or Repeated Start Bit (Sr), eventually releasing the bus wire.

TCP/IP

Sloppily referred to as the Internet Protocol, RFC 1122[42] is the industry standard Transmission Control Program, used by anyone connected to the internet since the wide-field adoption in the 1990's, when IBM and Xerox started adopted the protocol, nearly 20 years after its publication in 1974[17, 18].

OSI model

Layer	Name	Example protocols
7	Application Layer	HTTP, FTP, DNS, SNMP, Telnet
6	Presentation Layer	SSL, TLS
5	Session Layer	NetBIOS, PPTP
4	Transport Layer	TCP, UDP
3	Network Layer	IP, ARP, ICMP, IPSec
2	Data Link Layer	PPP, ATM, Ethernet
1	Physical Layer	Ethernet, USB, Bluetooth, IEEE802.11

Figure III[19]: The seven layers of the OSI-model. TCP/IP sits in layer 4. The IOTA Streams library also has a 3-layer model: Application, Transport, and Channels Layer.

TCP is structured as a four-layer protocol, with the lowest layer, the link layer, sitting at the transport level (layer 4) in the OSI-layer model. The now standalone IP protocol sits underneath the TCP protocol and communicates with the Data Link Layer.
The four layer are the Link Layer, the Internet Layer, the Transport Layer, and the Application Layer. The Application Layer can be compared to the top three layers of the OSI-model, and is the most outside-facing layer, providing data exchange and user interaction with data delivered by the lower layers. Basic Data channels are implemented by the Transport Layer with addressing, flow control, and multiplexing[20].

Channels inplemented by the Transport Layer function on router connection established by the Internet Layer, which's task consists of relaying data to the next router, bringing the packet closer to its destination. Finally, the Link Layer connects devices in the same network, providing the lowest means of data transfer inside the TCP protocol.

IOTA Streams

The IOTA Streams library is a messaging framework based on the IOTA DLT, aiming to provide crytographically secure message transferring on a scalable, reliable, and fast node network.

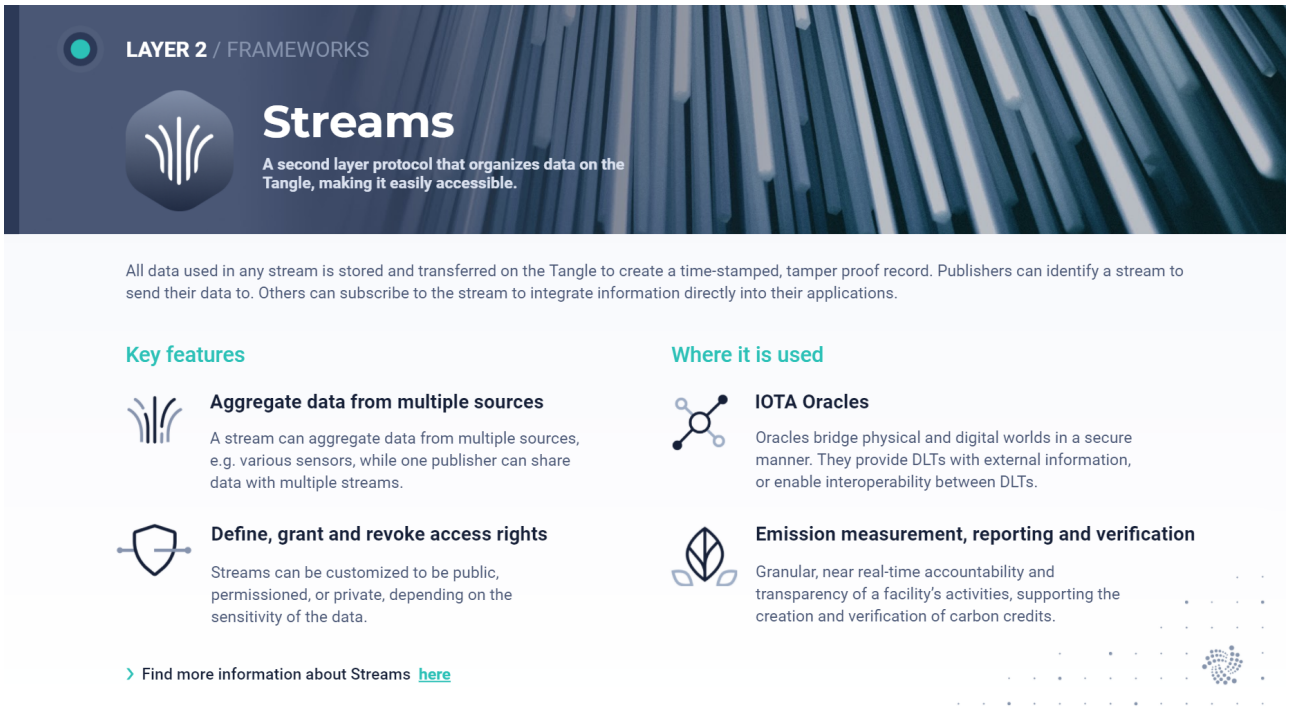

Figure IV[3]: Streams Overview panel depicting the place of the Streams-Protocol in the IOTA environment.

Leveraging the validating mechanisms discussed in *Chapter 1.2: IOTA*, the IOTA Streams library offers a handful of modules for en- and decoding, sequencing and sending of messages to the tangle. Namely, these are the application layer for developer interactions, the channels application, and the core layer, including the Keccak-F[1600] algorithm, the Ed25519 signature (as well as the X25519 key exchange), and the Data Definition and Manipulation Language (DDML)[44].

Spongos Automaton & Keccak-F[1600]

The Sponge function leveraged by the IOTA DLT to achieve cryptographical security is the Keccak-F[1600] algorithm, a permutation of the Keccak algorithm. The Keccak algorithm is a SHA-3 hashing algorithm, selected as winner of the Secure Hash Algorithm Competition by the National Institute od Standards and Technology (NIST) in 2012[21].

Keccak-F[1600] takes a bit rate of 1344 and a capacity of 256 bit to produce an output string of any length from the internal state of the sponge and an input string of any length[22]. Though still pseudo-random in nature, since the internal state of the machine cannot be accessed, one wouldn't be able to infer the output of the machine just by knowing the input.

More insight on state-of-the-art encryption algorithms can be found in the FIPS PUB 202 on the SHA-3 Standard[23] and the Keccak SHA-3 Submission[24].

The algorithm provides a strong enough cover against decoding, that current quantum computers would take unpractically long to crack such a hashed PRNG string. Currently announced are as many as 4,158 qubits in one system by IBM, expandable to as many as 16,632 qubits[25] as early as late 2023, though no clarification has been given, how many of these qubits are actually entangled or fully connected, a scalability criterium of the World Economic Forum (WEF). They report a maximum number of fully connected qubits of 24[26, p. 31].

As further stated by the WEF, current quantum systems fail on the task of scalability, with an estimated requirement of as much as 1,000,000 (1 M) qubits, "theoretically allowing for the creation of a sufficient number of error-corrected qubits to demonstrate quantum advantage in a real-life application"[26, p. 28].

Judging by comparison, we are in a healthy learning phase on our way to making quantum computing viable, but it will take much more time and research to actually provide applicable systems capable of cracking SHA-3 algorithms.

In addition, the NIST has been working on a post-quantum cryptographical standard since 2016[27], with the current current round 4 submissions announced in July 2022[28].

Framework Structure

In Streams, a channel can be formed with a number of authors and any number of subscribers. The author publishes messages in sequence, creating message links to refer to. The subscriber, if included in the subscription list called keyload, can read messages, with internal state catch-up mechanisms. A subscriber can be subscribed by transmitting his public key to the author, or by receiving a pre-shared key created by the author. Streams channels are non-fixed in type by default, meaning that a channel can have as many authors and subscribers at the same time. A channel can have multiple topics for the authors and subscribers to listen/read on. Each topic has its own permission settings, so that a User can be author in one channel, while being subscriber in different channels.

Messages are hashed and passed into an entry node and are then treated as any package on the tangle, being validated and publicly integrated into the network.

IOTA Wallet

The IOTA Wallet library provides means to create a seed, or root, from which any amount of accounts can be derived to house any amount of addresses. That gives enough flexibility to fit every use-case. In our case, we look for one seed, housing a number of accounts, of which at least one is accessible by the service providing water works. Remaining accounts should be connected to customers on the code side, serving as user interface to payment solutions.

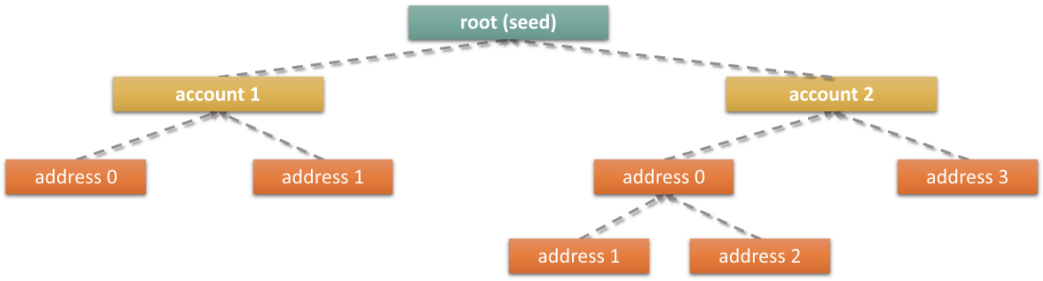

Figure V[36]: addresses are flexible to form any kind of structure. A single root approach increases safety of the funds itself, while many roots create a more distributed network harder to control.

FRAMEWORK STRUCTURE

Creating an account on the IOTA Testnet, as desired for a prototype, requires two managing entities, the Stronghold Secret Manager managing seed security and backups with the IOTA Stronghold library and the Account Manager. Stronghold is a separate library managing seed and key storage. It is a dependency of the Wallet libary making using it easy.

An account manager can create an account, that can create addresses. The account is connected to either the Mainnet, which currently is IOTA 1.5: Chrysalis, or can be connected to both Testnets of updates 1.5 or 2.0: Coordicide. We currently see no utility in using the 2.0 Testnet, so we connect to the 1.5 Testnet.

With an address, we can start sending and receiving funds. The wallet also offers a set of functions to interact with the account in different ways, e.g. fetching balance, listening on events and the like, which will become much more interesting in later development, when we start creating a GUI for the client side.

1.4. EXCURSION: WATER DEMAND & CONSUMPTION

As the product described later in this work is targeted at large water providers, it is a good idea to lay a cross-sector knowledge basis. This helps understanding the problem we're addressing and the potential value of the product beyond its monetary value.

Agriculture is the basis of many developing and emerging countries, and will always be an important industrial sector from a global point of view. It is vital, especially with the growing world population, to ensure access to fresh and clean water. This project aims at the demand side of the equation, offering the communication layer for a sensor-controlled network, but in abstraction, it can be applied to agricultural needs just as well to control water release based on temperature, ground humidity, sunlight intensity and air humidity. This chapter gives an excursus into the topic of agricultural water usage, giving an overview over the available scientific data, as well as insight into water usage globally, in Germany, and regionally.

WATER CONSUMPTION GLOBALLY

Water consumption varies greatly depending on the area, micro- and macro-climactic circumstances, as well as technical progress of the country.

Worldwide, about 70% of all available water resources go into agriculture[29]. In Europe, this number is only 24% combined[30], with southern countries having a higher percentage of up to 80%, when compared to northern countries.

Especially in Asia, over 2.000 km^3 of freshwater withdrawal (so called blue water sources) per year are used in agriculture, which is 81% of their total freshwater withdrawal, and amounts to 73% of the global annual freshwater withdrawal for agricultural use[31].

Unfortunately, only a third of the amount actually seeps into the ground. The rest either evaporates under high sunlight intensity and heat, or is washed away due to flooding which can happen when dry, hard ground cannot take water in as fast as it is flushed over.

According to a study from the EEA in 2009, the agricultural use of flood irrigation has been mostly replaced by droplet irrigation and underfloor irrigation in central Europe[32]. Which means, that on average, 8% of the water used in agricultural fields in Europe are prone to evaporation and flood.

That value is most likely to be higher in developing and emerging countries due to lower technical standards and lower availability of water distribution systems thus providing plenty of potential in the reduction of freshwater waste in agricultural fields.

WATER AS A REGIONAL RESOURCE IN GERMANY

To get a better sense of the current situation throughout Germany, we interviewed Mr. Robin Seifert from the Wasserwerke Zwickau GmbH (WWZ) on the topic of meter digitization in Germany and the catchment area of the company. He has been working for the company for five years and is the current project manager in the field sales & management. He is a former student from the Westsächsische Hochschule Zwickau (WHZ), having studied business engineering. Information presented in this section are taken from the interview.

INTRODUCTION

To give an overview, the WWZ maintain about 45,000 water meters serving around 200,000 inhabitants. This includes 17 cities and communities.

With roughly 2,200 meters being digital meters based on the wireless MBus-Interface, ca. 95% of all meters are static. 1,800 of the digital meters (81.82%) are installed in real estate owned by residential property companies. One such meter has a lifetime of six years, before its calibration invaldidates, while older models have a lifetime of one year. Both devices can receive a calibration extension which is tied to random sampling.

Users of the WWZ can be sorted in two categories, one being single- and duo- family properties, and the second one being apartment buildings. About 70% of the maintained properties are apartment buildings.

Aspects of Digitization

After introducing the current technological situation, we talked about advantages of digitization. Mr. Seifert showed great interest in distant reading and brought up the main points of interest a water provider has. Data reliability is the biggest interest in digitizing water meters, as water distribution is a payed service that needs billing. Billing is based off meter readings, which means the company requires current meter readings at least once a year. Currently, this reading is done by the user itself and by water works technicians only if a user misses to submit his data. With a distant reading method, both precision and reliability of the data can be ensured without needing to send technicians.

Another advantage for a water providing company would be a higher data density, allowing for economical planning throughout the year.

As far as realisation goes, Mr. Seifert expressed possible obstacles towards digitization in the transmission way from meter to a handling point for about 25% of the property in Germany are over 70 years old, and 41% over 40 years old[33], with the meters being positioned in cellars or otherwise obstructed in their transmission.

As to risks, data manipulation and data interference at or near the meter shouldn't be a problem, since the utilities of distant real-time readings have error margins. A large scale attack would be much more unpleasant, but a problem only with regards to data privacy. Much more volatile are centralized data gateways, like 3G-antennas. These antennas are out of reach of the meter data security measurements, and still will delay data transmission in case of interruptions.

To the advantages of a real-time reading, mostly quality-of-life criteria came up for water providers. Being able to map water feed and meter readings enables fast tracking of leakage and better preparation of demand peaks. The latter gets especially interesting when a large scale consumer as seated in the area, like large industry. In terms of water supply and large area statistical applications, even though a waterworks company may not be involved directly, Mr. Seifert still pointed out that a more sophisticated sensor network is likely to be necessary in case of extreme situations, like drought, fast and heavy climate changes, or a pandemic, as the last years have shown. In comment to the current situation, he said that an extreme situation might be necessary to demonstrate the demand for such a network. Currently, the effort exceeds the benefit.

Economic & Legal Aspects

To start of this section, a rough evaluation can be made based on the following data. To read out distant-reading meters, a technician has to drive through the city in a car. Though transmission range is up to one kilometre far, in some areas, especially in downtowns or rangy areas the range reduces as far as to single-digit metres, making a "drive-by" necessary. For the currently 2,200 meters a time-effort of 27-36 hours is needed, occupying two technicians and one car, which accumulates about 500 km of distance during the process. Mr. Seifert stated clearly that this must not be the final solution. In Denmark, a similar system exists, with antenna gateways mounted on garbage trucks. This ensures a regular reading of all meters, even in far remote residencies.

What holds many water providers back, is the legal situation around distant real-time meters in Germany.

Currently, European laws require residential property companies to switch to said meters until 2026. At the same time no federal law exists regulating the use of these meters. Under the aspect of the privacy act any one user in a single- or two-family property is allowed to refuse such a meter and delay the renovation process. Court decisions so far have been in favour of the water works company throughout Germany, but since 70% of the users live in such properties, companies fear the financial and timely effort lurking.
To end the section on a hopeful note, Mr. Seifert expressed his hope that a legal basis should be established in the near future, with pressure on authorities rising.

2. Economic Approach

Though this work is largely comprised of use-case oriented work from an engineering standpoint, a glance at economic requirements and possibilities is advantageous, especially under the premise, that this work serves as ground-work for a larger project. In this chapter, we present a model of business, formulating an existing problem, giving insight on how to solve it and the necessary steps towards that goal. We will see, that the market situation allow for bold movement, try to estimate a financial order of magnitude, in which we will be operating, and propose ideas for structure, partner networks, future development and infrastructure.
All information given in this chapter is reprocessed in a more extensive layout in our business plan, publicly disclosed until further notice.

2.1. The Current Situation & Problems

As of 2022, many Waterworks rely on the use of mechanical meters that require a technician to read the data off it. Since 2019 (WWZ, acc. to R. Seifert), waterworks have begun to exchange meters for modern meters that allow distant-reading and work on the M-Bus standard.
These meters provide a reasonable improvement and are worth installing but we think that we can improve on that, without loosing functionality, integrity, and staying in the same financial frame. Main critique points for modern meters are flexibility, data security, hardware security and troubleshooting.

2.2. A Possible Solution

We think, that the IOTA framework enables a higher security standard which will be mandatory in a few years to come. Our product allows not only for real-time data transmission but for real-time status control over the meter, the valve and the software. That way, troubleshooting can already be done from afar, before a technician is sent across the land for inspection. We maintain current standards in precision, utilize the same protocol and enable instant, bank-less transactions. Connected Databases allow for consumption peak mitigation and serve as regional, potentially country-wide real-time evolution map in water demand and consumption.

2.3. Scalability

While we will dive into technical aspects in *Chapter 3.1: Product Requirements*, this chapter gives an insight in what is to be done after this work with regards to building a serialized product able to be produced and distributed to larger markets.
We define three main categories, that need attention after the project has advanced from a prototype stage, with the first one being accessibility and flexibility, the second one being reliability, and lastly we take a look at adjusting the hardware to fit into an as small as possible production line.

Accessibility & Flexibility

Waterworks employ or commission technicians to install and put their meters into operation. This process should be shaped as straight forward, comprehensible as possible, with as little room for error as possible. Automation of this process is one very reasonable step towards this goal. We can automate most software commissioning with a program installed on a mobile device, such as a laptop or smartphone, only requiring the technician to start and monitor it.
To summarize, our client-side application needs to wait for an outside connection, ethernet or USB-cable springs to mind, verify that connection, establish a network connection to a provided server on the mobile device's side, exchange all necessary data needed to initiate connection to the server via the tangle and lastly verify that established connection by sending test messages.

In terms of flexibility, we aim for being able to install the product in as many environments as possible. Since we need an internet connection, we might struggle in old buildings, where radio connections of many sorts aren't viable. We think, that providing a split of functionality might be applicable here, with one device capturing data and controlling the status of the system, and one other device providing message and command forwarding. The connection between these is also the current approach for the prototype and realized via the I2C- and TCP/IP-Interface. Later, where necessary, we can replace the cable-driven connection for a radio connection, like Bluetooth, 868 MHz Bluebird or the like.
Flexibility is a large topic and ultimately, every single building is different, making this step one of the more resource-taxing, both in terms of time and Know-How.

Reliability

Since his project currently is in prototype stage and will advance to a field-testing stage before finally being fully released, it is best to secure the transfer of the critical consumption data that we handle. We think, that this is best achieved through establishing a conventional transfer line to the waterwork's server. Though it might not be as secure, for a field-test it serves the valuable purpose of confirming data sent through the tangle. It also serves as fallback, in case anything tremendous happens to the main network. We are handling critical information regarding a basic human right, so it is best to maintain a necessary level of resistance.

For us, full control over the network status is very important. We want to be able to know any error or unexpected behaviour, without needing to inspect the device on-site. We already forward every error possible to the server-sided program, but only a fraction actually get handled. Error handling and fallback solutions greatly exceed this work's scope, but we want to emphasize their importance in this section.

Making a software-hardware combination future-proof includes making it upgradeable from afar. Ideally, since we already have an internet connection, we can leverage that connection to download updates from the server automatically.

HARDWARE ADJUSTMENTS

Currently, the prototype runs on a Raspberry Pi. Though a reliable product in a prototype setting, the Raspberry Pi offers too much functionality that is not needed in a serialized product. Getting rid of it and replacing it with another or a custom board is a valuable step, as it lowers production cost greatly.

Combining all three hardware products, the meter, the valve, and the circuit board, into one single physical unit makes production a lot easier and faster, so we aim for a partnership with water meter manufacturers, to create such product. This work could be done with the claim of a state subsidy, like the ZIM funding for medium-sized enterprises, either on ours or the partner's side.

2.4. FOUNDING

The current status of the project makes it eligible for the german founder scholarship "EXIST". The program is designed for a team of one to three students or graduates looking to develop an existing technology into a defined product. Since a prototype will be made available during this work, an application seems appropriate. The program covers salary for every team member for up to 12 months, as high as 2,500 € for graduates. It also includes a budget of 30,000 € for material costs and services and 5,000 € for coaching services. For a 12 month period, this roughly estimates to a total sum of 119,000 € for a team of two graduates and a technical associate.
A more detailed breakdown of both the immediate costs during the first year of scholarship and a long-term estimate for running a net-positive business can be found in *Chapter 2.5, Budgetary Planning*.

2.5. BUSINESS INSIGHTS

ECONOMIC CONCEPT

Economic efficiency for this project derives from not only the finesse funnelled into the technical product. The product also includes software, an API and the financial services connected to the payment functionality. Once the technical product with all its advantages has been established, software is rented in a Software-as-a-Service (SaaS) -model, allowing further improvements and expanding the functionality. This binds customers and enables more directed feedback and improvements, providing profit for both sides.
In a best case scenario hard- and software are married in a single physical product. This leads to a monopoly position, the receiving partner being forced into a situation where they rely on the quality of service and goods of one company. There are, on a side-note, ways the circumvent and prevent this by encouraging competition, licensing agreements, and even (financially) supporting competition. More on that topic can be read in the corresponding business plan. For now, this monopoly position provides a great opportunity for both customer binding and responsibility.
In the last step of business development, providing financial services can bring great success. Though a rather large project financially, a service fee of less than one percent can already generate large amounts of

cash flow. This is charged for providing enough of both FIAT- and cryptocurrency to exchange. The IOTA tokens are used as financial asset with which the consumer can pay monthly bills or charges in almost real-time, with no need for an intermediary, like a bank.

MODULAR APPROACH

Different customers and users require different software packages and have different priorities for functionality. As modern software provider, we want to be able to tailor our software to our users needs, while still being able to target a large bandwidth of customers. A modular approach seems the most fitting, unifying flexibility and reliability and defining the skill set that we want to bring to the market.

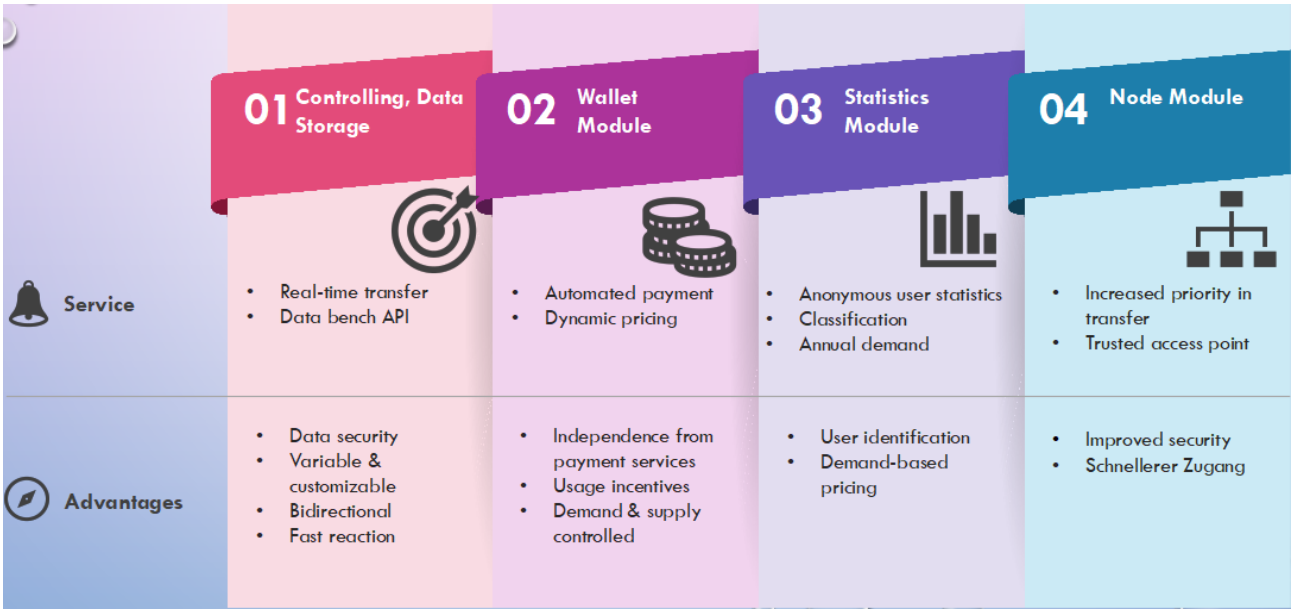

Figure VI: A modular approach to SaaS marries flexibility for customer needs and easy-to-expand software architecture. Image provided by C. Butze

As our core project, we focus on stability, security, and efficiency of the data transmission. The better our performance in this core module is, the higher standard we set for our customers and users, and the better our side modules are going to work. These side modules are the wallet module, which we include in the prototype of this work already, but plan to separate soon due to its specific use-case, the statistics module, enabling large scale empirical data processing, especially interesting for scientific and governmental use, and the node module, which comes with access to a high computing power access node maintained by us, as secure and fast access point for our customers that don't want to rely on public entrance nodes.

VALUE-ADDED PROCESS

To give an overview over what services and products bring the company's value, we try to isolate them in this chapter.

Hardware.

The hardware of finished product can be divided into four distinct parts, the water meter providing data, a

valve for flow control, a hardware PIC acting as executing controller between the two former devices and the fourth part, the software housing hardware.
All of these parts could and should be married into a single physical product, ideally in partnership with water meter manufacturers. This then would add to the value-added process, though isn't an immediate process.
The water meter and valve are an existing technology not part of the value-added process in this project. Manufacturing and calibrations extend the scope of this project and should be outsourced.
The aforementioned controller, in short PIC-circuit, leverages I2C and M-Bus technology, as well as microcontrollers, to access the water meters data and control valve position and feedback. It is connected to the client-sided software as a I2C-slave. In addition to controlling movement of the valve and reading data from the meter, it also gives detailed status information about both devices, that are forwarded to the server for further processing.

Client-Sided Software.

To define delivered value in software, we have to differ between young and old software. While young software consists of research & development, tailoring, and surveys, Existing software has more to do with customer and user support as well as bug-fixing. Today, many application run on a SaaS-model which allows both stages of the product to be maintained.
In this work, we present the general technical functions and possibilities of the underlying framework. The framework itself isn't our doing. While doing so, we leverage several existing protocols, like the interfaces presented in *Chapter 1.3: Leveraged Interfaces* or the Raspberry Pi, serving as housing unit for our software. The concept that we produced consists of many functionalities, that we develop ourself.
The client-side software will be delivered with the hardware product and therefore charged once.

Server-Sided Software.

The server-sided Software is much less dependant on underlying hardware than the client side. In this work, we establish a connection to the tangle, a simple pricing algorithm, an API to databases, and a wallet structure, enabling transfers to a user account. This software is highly customizable and subject to constant change, which makes it perfect to rent in a SaaS-model.

MARKET & COMPETITION

The target market we aim to operate is a narrow market with slow changes that is extremely dependant on laws and external investments. Communal companies aren't necessarily expected to run net-positive, but are certainly under close watch by official institutions. Still, they're financed by the tax-payer.

CRYPTO TECHNOLOGY

Crypto technology and especially crypto security are rather new fields just recently getting attention by larger companies like IBM, American Express, Bosch, T-mobile, Siemens, Visa and many more[34].

Expertise in the field is rare, and an opportunity can be taken to establish broad presence in the market early-on. That also means that Know-How isn't broadly available. Partnership with a university is most aspired to. Scepticism of potential investors can be relieved by a clear, up to academical standards documentation, which this work aims to deliver.

TARGET MARKET

The product shown in this work aims at B2B-customers. Though our users are private households, The Waterworks decide about the meter to be used and are therefore our direct paying customers. Distribution to private households is not intended. A possible target is the political landscape on communal and regional level, to inspire a legal basis, on which meters like the one presented in this work are required. This would also encourage competition in the market which would fuel research and development, improving user experiences.

MARKETING & SALES

As in every upcoming product, our project is only worth as much as someone is willing to pay it attention to. To propagate the project and product we opt for personal Marketing or Networking. Management budget is both not available and not necessary in the water sector. Since the waterworks decide what meters they use, user binding and direct marketing to private households would be redundant. That on the other hand allows us to focus on B2B-connections, building a partner network and tailoring our software solutions to the needs of our customers.
Part of our marketing strategy is a website. It is mandatory for success nowadays and will be done. Later, the website can serve as recruiting platform, public announcement channel, as well as to establish a corporate identity.

ORGANIZATION

Planning the structure of a company early-on is both hard work and rewarding in the long run. Establishing hierarchies and structural units is crucial in management and productivity. However there is much difference in young and established companies, especially when it comes to distributing work among employees.

In our early days, a flat hierarchy with short communication channels is desired, as there will be a lot of work to do that just doesn't fit into job descriptions. Employees should be motivated and willing to do tasks that are not directly their field of work.
Later on, it is indispensable to create multiple structural units that each have their own separate field of work. We have outlined here a simplified organigram showing the main fields.

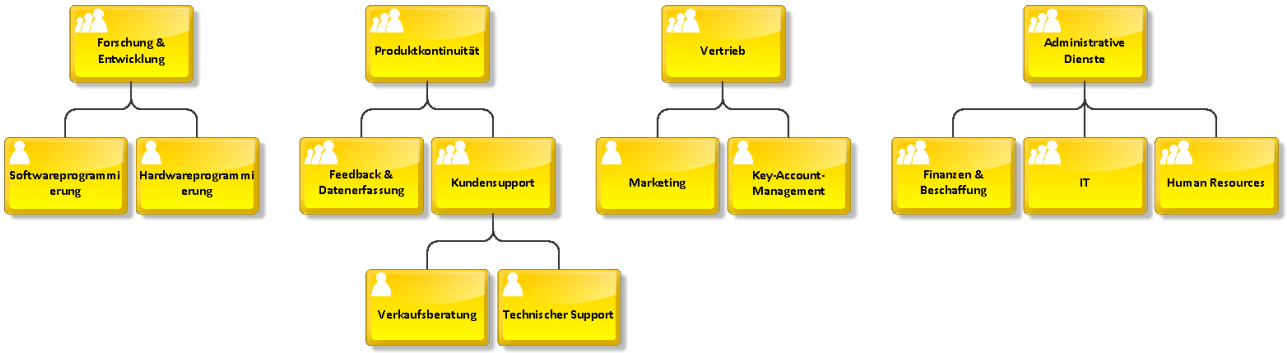

Figure VII: Simplified organigram showing the first expansion

BUDGETARY PLANNING

Budgetary Planning is conceptually divided into two segments, of which the first one focuses on optimizing expenses during the scholarship, and the second one focuses on an estimated financial income statement. This subdivision is necessary in our eyes, due to the limited scholarship duration and a smooth transition between scholarship, which includes mainly research and development, and building a commercially functional company.

The main difference between both is the expenses in material and production cost. During the first year of development, a modicum of expenses will go to basic equipment, wages, and services. The EXIST-program allows team members to access the partnering university's infrastructure and work places freely, meaning expenses will be limited in regards to office equipment.

Pos.	Name	Description	Amount	Cost	Sum
	Salaries			**per month**	**per year**
1	Salary Graduate		2	2,500 €	60,000 €
2	Salary techn. associate		1	2,000 €	24,000 €
	Coaching			**once**	**once**
1	Notary costs		1	500 €	500 €
2	legal advice - founding		1	1,500 €	1,500 €
3	legal advice - patent		1	1,500 €	1,500 €
4	general legal advice		1	1,500 €	1,500 €
	Material Costs				
1	Office equipment	• mobile computer (x1) • software licenses • server cost	3	120 €	860 €
	Service Costs				
1	Hardware circuit	commissioned to external	1	5,000 €	5,000€
2	External services	Software development Software audits	1	5,000 €	5,000 €
3	Rent and catering	During working time at Zwickau, an apartment will be rented	12	800 €	9,600 €
5	Travel expenses - fix	Travel expenses between Freital and Zwickau	1	1,000 €	1,000 €
6	Other travel expenses	Hotel and catering for possible meetings	1	2,500 €	2,500 €
7	Marketing and Networking	• Flyers • meeting rooms • presentations • website	1	5,000 €	5,000 €
8	Communication	compensation for use of private devices and services	3	50 €	150 €
	Total				***29,110 €***

Figure VIII: Early budget planning for the EXIST scholarship program

Expenses after a serialized product has been developed can only be estimated. The following approach should be seen as early approximation.
It includes a rough overview over one-time expenses for notary costs for establishing the company, other legal services, office equipment, technical equipment, most necessary sanitary equipment, artisan services, basic marketing expenses, and stock goods. After that, estimated current expenses for operational expenses, production, and salaries are given.

ONE-TIME EXPENSES

Position	Description	Sum
Founding Expenses		
Legal advice		1,500 €
Records	Trade register	300 €
External services	Notary, broker	5,000 €
Long-term Investments		
Office furniture		1,400 €
Electronic equipment (except measuring equipment)	Computer, Windows licenses, land-line, printer, Server equipment	4,100 €
Measuring equipment		500 €
Office products		300 €
Kitchen and sanitary equipment	Refrigerator, microwave, silverware and dishes, soap, towels	575 €
Cleaning agents	vacuum cleaner, broom, mop, towels, cleaning agents	150 €
External Services	Painter, artisans, electricians, etc.	up to 7,500 €
Miscellaneous	Window sticker, plants, door label	600 €
Short-term Investments		
First purchase stock ware	Card board, printing paper, shipping labels	300 €
Deposit		2,000 €

Sum: ***24,225.00 €***

Figure IX: Estimated one time expenses for working stations

BUDGETING

Data referring to one year of production, 5,000 pcs.

Position	Description	Sum
Operational Expenses		
Software licenses / business portal	Office software, Adobe, Firewalls, individual requirements	3,600 €
Land-line, power, internet	600 € Internet & land line 2.000 € Power	2,600 €
Rent		42,000 €
Insurance		200 €
Production Cost		
Water Meter	ca. 40 € p.P., includes calibration	200,000 €
Valve	ca. 8 € p.P.	40,000 €
Controller circuit	ca. 2 € p.P.	10,000 €
Main circuit	ca. 10 € p.P.	50,000 €
Salaries		
Hardware engineer, techn. advice		35,000 €
Software engineer, techn. advice		30,000 €
Accounting / IT	External services	10,000 €
CI, care for business partners, networking, HR, project- and time management		24,000 €

Sum: ***447,400.00 €***

Figure X: Estimated regular expenses

RISK ANALYSIS

As high as the chances in crypto technology are, as is the lack of qualified engineers affine to the field. Risk is high to hire someone not suited for the project or to propagate wrong expectations.

Crypto technology is a field seen as highly sceptical by a wide spectrum of people. A gapless documentation and high transparency is needed to clear the fog with facts.

The water meter market is a slow one, even though it is one with monopoly status and large unit sizes. Also the projects competes with “standard” distant reading meters that become more and more prominent throughout the market.

Last but not least, a look onto the current global situation. War in Europe and delivery problems in the far east have thrown shadows over the economy. Should chinese companies be considered for production, the current geo-political and geo-economical situation needs evaluation.

PROSPECT

We hope that this chapter was able to give enough insight into the economical aspect of the project as well as the fact that we are well aware of both risks and opportunities to present a clear view on the future actions regarding the success of this project. This work delivers a basis from which we aim to improve on systems and structure to produce a product up to the standards of our customers and users.

Experience from this work can be applied in later projects, when research is done in neighbouring fields and techniques, such as agriculture, building services, sensor technology in ventilation technology, early alarm systems or sustainability.
Especially excited is the upcoming Bachelor Thesis by Mr. Butze, a fellow colleague, on the topic "(Development of a SaaS Business Model based on the IOTA Ledger Technology using the example of a radon sensor network" (original title: "Erabeitung eines SaaS Geschäftsmodells auf Basis der IOTA-Ledger Technologie am Beispiel eines Radonsensornetzwerkes").

3. Technical Conceptualization

In this chapter we come back to the technical side of this work and show how the technologies presented in *Chapter 1: Theory* can be combined and formed into a product. We translate economical requirements into hard- and software requirements to formulate steps towards a working prototype.

3.1. Product Requirements

Large project management requires narrow conditions to ensure a focused, expedient work flow. Main functionality is pinned down fast, but frameworks and optional features beyond pure functionality need careful consideration, especially, when working under limiting conditions, such as time and budget.
In the following, we list what we find to be the most necessary functions the product needs to execute, and later design a product that include additional features enabling or paving the way for a more sophisticated later version of the product.

Core Functionality

As mentioned in the abstract, the product should fulfil three main functionalities. Mapping data to enable real-time resource management, incentivise consumption, and providing payment services for customers.

Each functionality leverages a different software framework or hardware protocol as can be seen in Figure XI.

Data mapping	Usage incentives	Payment Service
IOTA Streams, MBus, TCP/IP, I2C	RuSQLite, TCP/IP, I2C	IOTA Wallet
Bi-directional communication layer real-time data provisioning	Pricing algorithms command-issuing remote-controlled ball valve	real-time payments highly secure, decentralized payment structure

Figure XI: Overview over the core functionalities of the prototype: data mapping, usage incentives, and payment services.

Data Mapping

The main usage of a water meter is to map the water consumption of the pipe system it is attached to. Industry-standard water meters with electrical Interface currently run either MBus- or ModBus-Interfaces. Since we will use MBus as discussed in *Chapter 1.3: Leveraged Interfaces*, we need a MBus controller acting as a master to the water meter in order to access the data.
The gathered data from the water meter now needs to be sent to a server-sided application for further logical processing. This is done with the IOTA Streams library for Rust, where cryptographic concepts are leveraged to secure the data send.
IOTA Streams has a Author - Subscriber structure with different access models. Since we need a bi-directional communication, a multi-branch structure[1], where both participants listen to each other seems to be best suited. It removes the necessity of sequencing messages while still allowing for the maximum security in data transfer.

Usage Incentives

Water is, apart from pure oxygen, the most important resource for humans on this planet. Not only do we consist mainly of it and need it to survive, it is also the basis of food for all crops and animals that fall into the food pyramid and the sensitive ecosystem of life.
A large part of this project is to enable intelligent resource management, which not only includes mapping water consumption and demand over time, but also being able to react to high demand phases like a wildfire or to prevent droughts. To realize this, the product includes a remote-controlled ball valve. It is mainly implemented to allow water to be turned off from afar, making it a much more comfortable, cost-friendly solution, and also paves way to deal with non-paying customers and gives the ultimate tool for emergency situations.
To further incentivise customers water consumption, an underlying pricing algorithm should intake collected data from a data bench and other (regional) data sources to vary prices depending on the mid- to long term demand, like in a drought, or during especially hot summers. This is very interesting for agricultural fields where farmers should be incentivised to automate sprinkling based on humidity and intensity of sunlight. This way, a thoughtful and sustainable handling of the resource can be incentivized.

Payment Services

To deliver a full product and enable the full potential of cryptographic frameworks, we implement the IOTA Wallet library. It creates users on the blockchain-like network and assigns accounts and wallets for currency management. One main account with many wallets is well suited for the purpose of this work, as it simplifies payments made by users. Users can acquire currency and use it for all necessary services related to their waterwork.
These service are validated in seconds and cryptographically secured, improving both customer and user experience. With the elimination of an intermediary without diminishing payment security and integrity customers can take their financial matters in their own hands, while providing quality of live increase to their

users.

3.2. TIME-LINING

To narrow down the necessary steps to a working prototype, the project is split into three modules with distinct applications and functionality. Two of these modules are subject of this work, with the third one being designed and assembled by Dipl. Ing. Christian Flegel.

METER MODULE

1. Sends data gathered from the PIC and sends it to the tangle periodically
    - Consumption Data
    - Status Data
    - Errors
2. Receives data and commands from the tangle
    - Pricing Data
    - Valve Commands
3. Displays useful information like consumed amount (total/current interval), balance, estimated remaining amount

SERVER MODULE

1. Receives consumption, status, and error data from the tangle
2. Sends commands and pricing data to the tangle
3. Receives payments from the client
4. Stores data to be transferred into an attached data bench
5. Extracts data from data bench
6. Calculates pricing data, commands, large scale statistics

CONTROL BOARD / PIC

1. Serves as MBus master to the Water meter to extract consumption data
2. Serves as I2C-Slave to present data to the Meter Module
3. Serves as controller for the ball valve to execute commands

To implement the functionality presented above, a time-line has been fleshed out.

STEP 1 - DATA STREAM

1. Select a water meter to use in the project, connect it to a Raspberry Pi over the MBus-Interface
2. Write transmission behaviour for two Raspberry Pis

2.1. I2C-connection

2.2. Tangle-connection

2.3. TCP Network connection

Step 2 - Commands

1. Serialize command structure, identify necessary commands
2. Connect ball valve with electrical steering
3. Forward commands to the PIC-Circuit

Step 3 - Error Handling

1. Serialize Error structure, identify necessary errors
2. Forward errors from the PIC-Circuit

Step 4 - Data storage

1. Set up data bench
2. Format incoming data and store inside data bench

Step 5 - Data calculation

1. Extract data from data bench
2. Simple pricing algorithm to demonstrate possibilities
3. Simple statistical evaluations to demonstrate possibilities

Step 6 - Command Line User Interface

1. Set up a simple user interface displaying information on the Client Side

4. The Prototype

After we have successfully layed out a roadmap to follow while we design our code, we can now choose the hardware components that best suit our needs. In this chapter we present the components and their associated functions in their context.

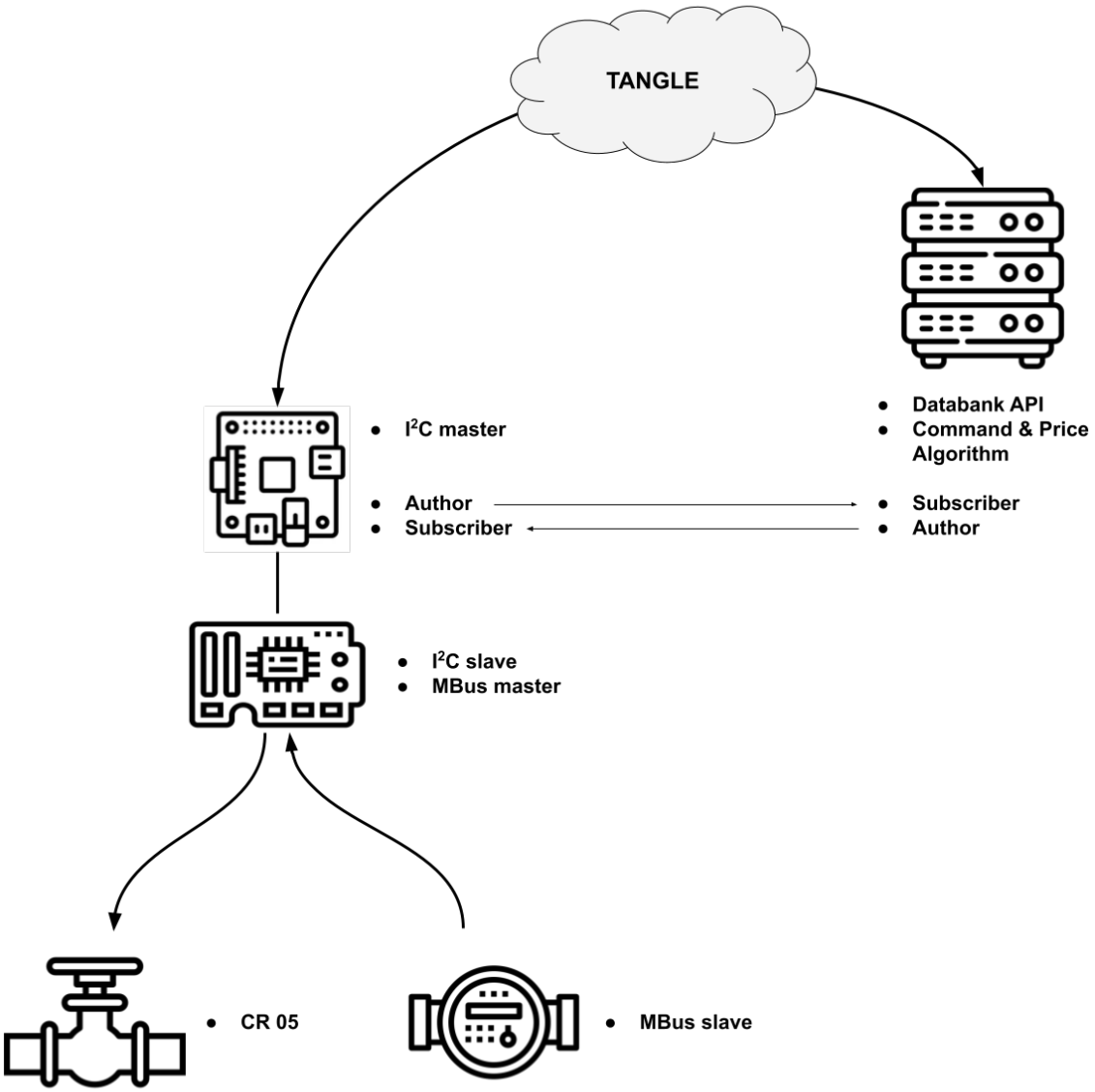

Figure XII: Visual representation of the conceived hardware composition. Functional deferrals are represented in outsourcing the lower-level communication to the PIC-circuit.

4.1. Water Meter

Choosing a water meter for the project, we narrowed down on MBus water meters that provide a hardware interface for us to read out. Lesser focus is layed onto the measurement method. The meter used is an impeller counter, one of the lesser sophisticated methods, widely used in end user environments.

The type description of the meter is as follows:

GSD8-RFM by Bmeters Srl Italy

The meter provides internal data in the form of an MBus interface, acting as the MBus slave[35]. We can poll data from the meter sending commands as a MBus master. Due to the nature of the MBus protocol, data can be requested in real time, allowing for a real-time control over data and status.

4.2. Electrical Ball Valve

As valve we chose to go with a CR 05 type electrical ball valve. This type of wiring allows for full oversight of the status of the valve as well as the integrity of the physical circuit. Three control wires give out signals according to movement and position of the valve and if all cables are working as intended.

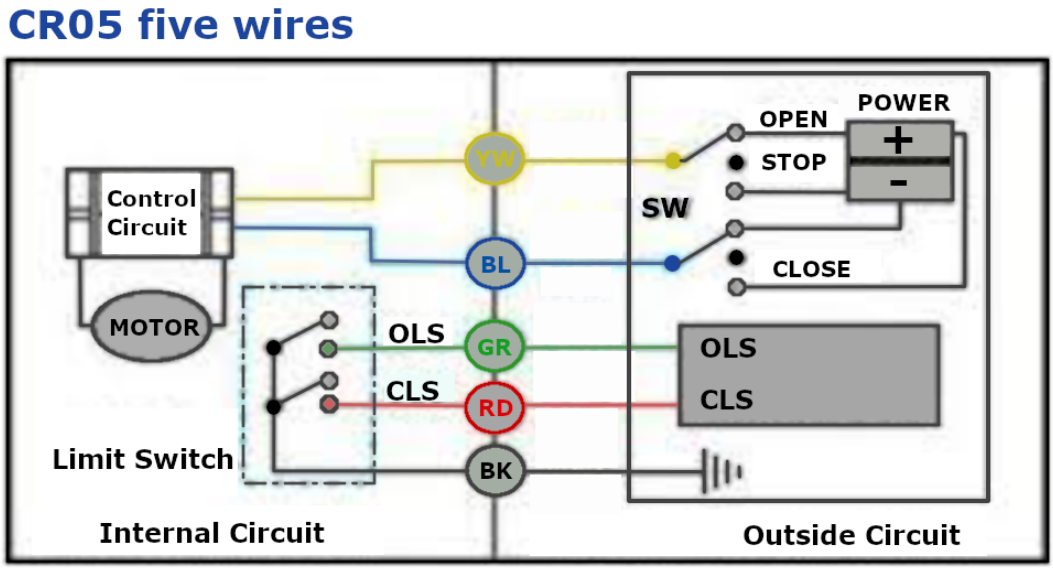

Figure XIII: CR 05 wiring diagram. Lines YW and BL allow status detection of the current valve position, while the OLS and CLS provide status confirmation and connection detection. Courtesy of Shenzhen Kangmwang Electric Co,., Ltd., edited for higher resolution by F. Rosenkranz

With this wiring, we can derive all necessary signals for controlling. For basic motor control, the upper lines coded in yellow and blue transmit intended behaviour to the internal motor. For control, the lower three lines coded in green, red, and black signal an existing connection (BK), as well as the current status of the limit switch (GR, RD). Through software logic, Dipl. Ing. C. Flegel was able to make the following statuses and errors available for forwarding on the PIC-circuit:

► Open ► Opening ► Close ► Closing

for identifying the current position,

► Undefined

as catch-all fallback status,

► Motor Blocked ► Motor not working as intended ► Motor has over-current

in case of systemic events, and

► Motor error

as catch-all fallback for unintended behaviour. *Chapter 5:Application Code* and *App. A: Full Program Code* go into more detail, how this information is handled on both Client and Server side.

4.3. Raspberry Pi

To house the software for this project, two Raspberry Pis have been deemed suitable. The thought process behind choosing the Pis is the following:

- Both Pis have been used to represent a Client-Server-structure before[1]
- That structure leveraged the IOTA-Streams library to prove a similar concept
- A Raspberry Pi is transportable
- A Raspberry Pi allows physical distinction between software (vs. virtual machines)
- Raspberry Pis allow the user to access peripheral interfaces easier than windows-driven tower PCs

The task of the raspberry is to connect to the PIC via the I2C-interface to poll data and issue commands, as well as connecting to the tangle in order to send gathered data to the server and await commands.
We've installed Ubuntu 20.10 on the Pis and updated to 21.04.01 LTS about halfway into this work. Ubuntu seemed to be a combination of both easily configurable and easy to learn due to it being open-source. Programming has been done on windows, as experience with other operating systems was scarce and it seemed most convenient.

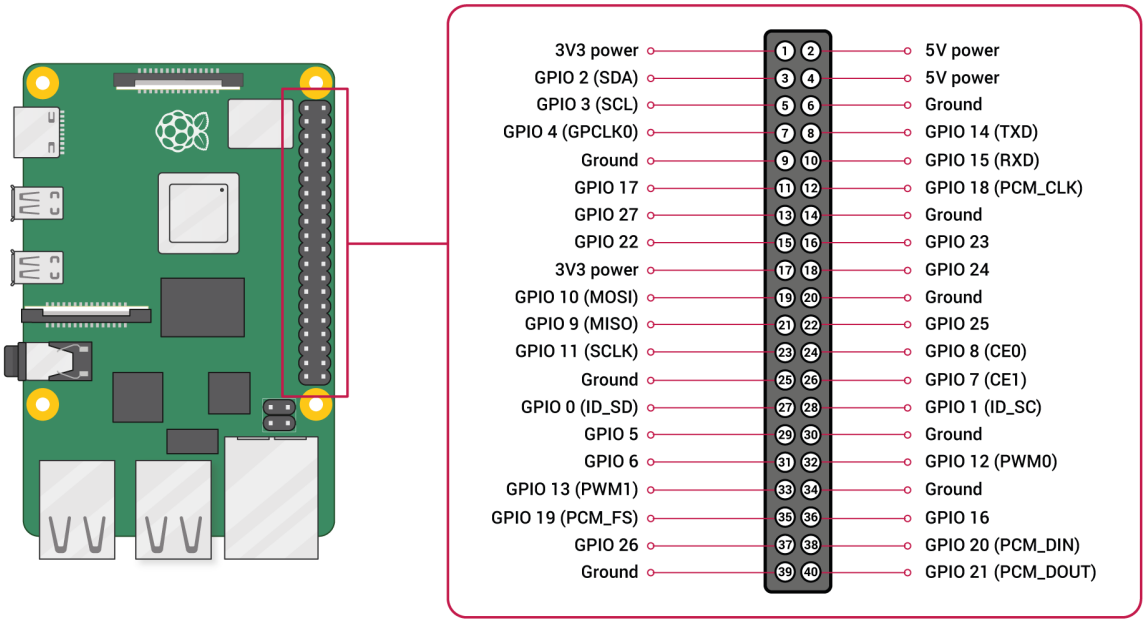

Figure XIV[37]: Raspberry Pi 3 Model B+ pin layout. PP numbers can be found in the middle.

GPIO-Pins & I2C-Interface

The Raspberry Pi 4 possesses a range of hardware interfaces available for leverage by the user. The GPIO-pins provide a base line hardware interface with several 3V3 and Ground pins. Two 5V pins are accessible, tho it should be noted that none of the input/output pins on the board actually support 5V DC, as it will physically harm or destroy them.

Pull-up and pull-down resistors are fixed on physical pins (PP) nr. 03 and 05, corresponding to GPIO02 and GPIO03, while other pins can be configured to provide either or none while unconnected. Due to the pins being rather baseline in functionality, a range of peripherals can be connected and communicated with, including pulse width modulation (PWM) on PP 12, 32, and 33, Serial pins and I2C on PP 03, 05, 27, and 28. It is important to note, that throughout this work, we are referring to the pins with their physical number, not their GPIO-nr. to avoid confusion. For example, GPIO pin nr. 22 sits on the PP 15, we refer to it as nr. 15.

4.4. PIC-Circuit

In order to access the water meter and the valve from a single access point, with lower level (hardware-embedded) programming, Dipl. Ing. C. Flegel was tasked with designing a PIC-circuit that acts as MBus-master to the meter and refines commands from a controlling software into signals readable by the valve. In order to access the PIC, we opted to leverage the I2C-Interface available on the Raspberry Pi, requiring the PIC to also act as I2C-slave.

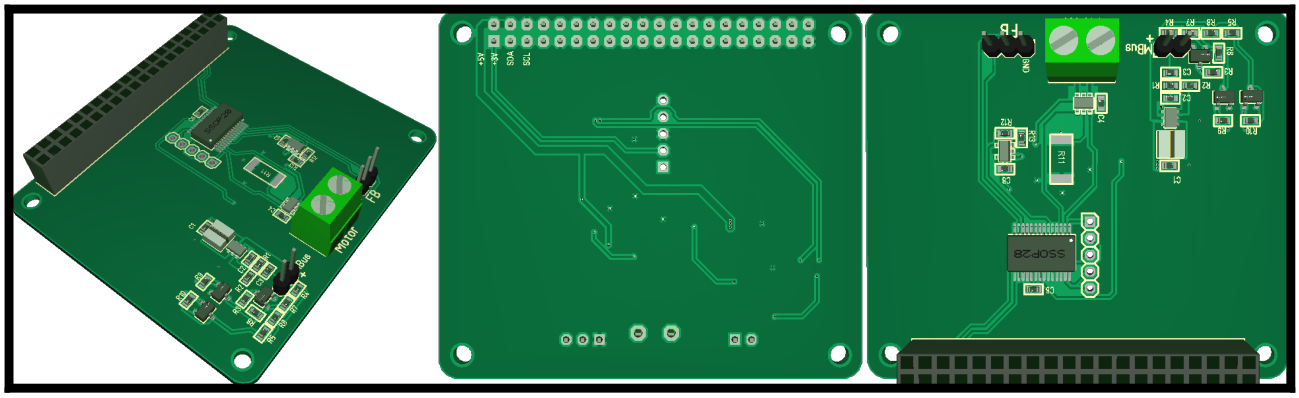

Figure XV: PIC-Circuit design shots mid-development. From left to right: isometric view, bottom view, top view
Designed by Dipl. Ing. C. Flegel

Connecting our PIC-circuit to the raspberry is done by a matching pin header, as seen in Figure XV and XVI. This brings both short connections and stability to the structure. The Pi provides 3V3 and 5V voltage to the board via PP 01 and 02, respectively.

5V voltage is required for three of the five components on the board, namely the step-up converter, the motor control, and the operation amplifier.

The latter one taps the motor control wire and amplifies its signal to the PIC microcontroller for monitoring, while the first one converts the supplied 5V voltage to the MBus bus idle niveau of 36V for this component.

The only component utilising the Raspberri's native 3V3 power supply is the PIC microcontroller itself. The PIC-board is grounded all six available pins on the Raspberry Pi.

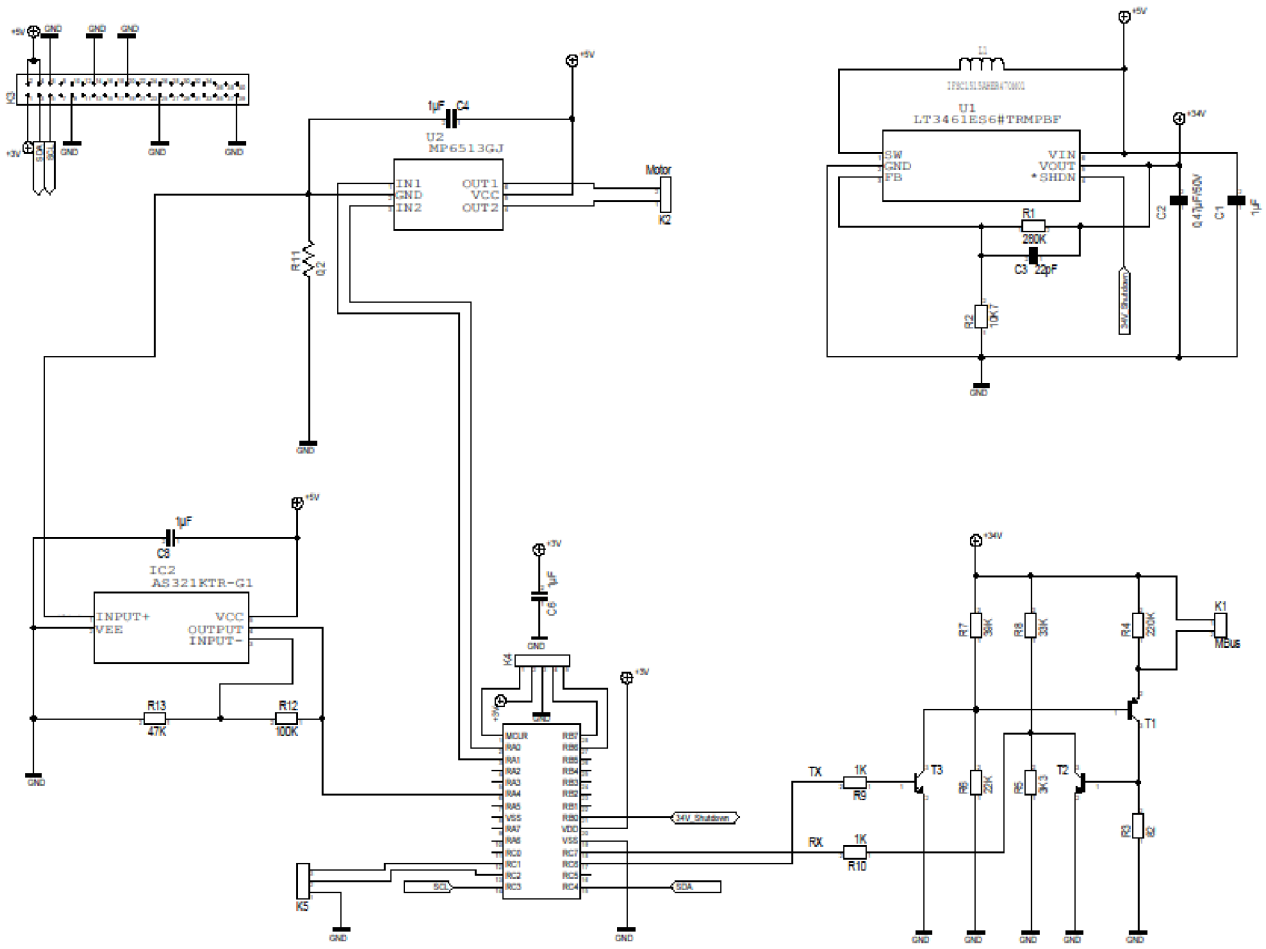

Figure XVI: Circuit diagram of the PIC-Circuit. Beginning from the left, going clock-wise, we can see the Op. amplifier, the motor control, the step-up converter, the MBus physical components, and the microcontroller.

Since the data sequencing of Serial UART is identical to the MBus sequencing, Mr. Flegel was able to leverage the PIC's physical UART-interface of the microcontroller. As microcontroller, he chose the PIC16F18054 by Microchip Technologies Inc.

5. Application Code

This Chapter takes a closer look onto the overlaying software application that aims to provide the required functionality on the client-sided device. To make this chapter as easily digestible as possible, we will look at the basic layout of our code and then at leveraged functionalities of the code in very specific settings, with shortened examples. A full program text of both the client-side and the server-side can be found in *Appendix A: Full Program Code*.

Where applicable, we include comments to further work and hint at the current weaknesses of the product.

5.1. Overview

To give a small overview over the Rust programming language, we start the project by calling the rust built-in crate manager cargo:

§~: cargo new kaskade

which creates a working directory for us. Cargo also allows us to include external libraries, crates, which we specify inside the Cargo.toml:

Cargo.toml

```
---snip---
[dependencies]
anyhow = { version = "1.0", default-features = false }
streams = { git = "https://github.com/iotaledger/streams", branch = "develop" }[1]
rand = "0.8.4"
rppal = "0.13.1"
thiserror = "1.0"
tokio = { version = "1.5.0", features = ["full"] }
```

Note the **rppal** crate. This crate only compiles on a Raspberry Pi, due to its native hardware environment. To develop on a Windows, Linux or WSL system, one should consider either directly developing on the Pi, using cross-compilation, or leveraging SSH-tools like PuTTY or Visual Studio Code's Remote Explorer, as we did here.

We will go into detail about how each of these crates add value to the code at the respective points, for now some additional organization takes place. Rust projects allow modulation and organization with the use of the **mod** keyword. Inside lib.rs, we can declare modules and make them available for use throughout the whole projects with **pub**:

[1] On a small sidenote: the IOTA Streams Library has changed namespaces drastically throughout the development of this project. The iota-streams crate is now called just streams, which might cause confusion with other messaging frameworks.

lib.rs

```
pub mod error;
pub mod i2c;
pub mod streams;
pub mod tcp;
pub mod threads;
pub mod utils;
```

Here is the whole working directory structure:

root/	
|-- src/	
|-- backups/	
|-- author.backup	encoded backup of the author
\-- subscriber.backup	encoded backup of the subscriber
|-- error.rs	error definition, trait declaration
|-- i2c.rs	data-requesting, command-forwarding to the PIC-circuit
|-- lib.rs	module definition and management
|-- main.rs	main program code
|-- streams.rs	crypto-messaging management
|-- tcp.rs	pre-tangle data transmission; establishing a tangle stream
|-- threads.rs	basic thread management
\-- utils.rs	utility functions without assignments
\-- Cargo.toml	Crate management file

5.2. THREADS & CONCURRENCY

In order to understand the necessity of intra-program data management, we find it important to look at two things: data ownership in Rust and the IOTA Streams library.

In Rust, a variable can only ever be owned by a single owner, or live in a single scope. An owner is a named variable that holds a value, and a scope is the part of a program in which an item is valid[38]. At the end of a scope, or when a value is moved outside the current scope, calling that value will lead to a compile-time error.

Rusts way of allocating data to either the heap or the stack usually puts these kind of processes into the background. It helps, that types whose size is known at compile-time automatically implement the Copy trait, allowing us to call them over and over again inside the same scope.

Unfortunately, the IOTA Streams **User<Client>** type doesn't implement the copy trait. If we want our program to run multiple tasks in parallel, and we want to, then we need to have a way for our variables and functions to access the data concurrently.

PARALLELISM

We want our programs to be parallel, as soon as they have multiple super-positioned input/output functions running. We want to poll data via I2C at the same time as we want to send and receive data to and from the server. We cannot delay one for the other, like we would in a linear program. One error, misplaced cable, over current, or simply a malfunctioning transistor should never cause the whole system to grind to a halt.

MUTABLE REFERENCES AND THE TOKIO CRATE

To make sure no one can edit our unknown-sized value while someone else is reading it, we have two options, either locking the value and only handing out one key, or creating a code structure that allows the value being moved safely and no one requesting that value before the value returns. Using an **RwLock**, as one would think, isn't actually necessary, when we make one trade-off.
That trade-off is, that we can only send one value to the tangle at a time. This is a discussion to be had, when this work goes into its refinement phase, but for now, we are actually fine by creating a simple line of messages. To further emphasize this, we group data instead of sending every status, data, and error separately, which would at least favour a parallel behaviour of the tangle output.
Having made that decision, we can pass mutable references containing the **User<Client>** around in one thread, while other threads take care of the data polling, command forwarding and so on.

The tokio crate offers an improved version of multi-threading. For further detail, the tokio book[39] and the tokio docs[40] are great resources to get started. The IOTA Streams library depends on the crate, so it is no effort to include it into our work as well.

Now, we can dive into the different parts of the program. And we will do so by following two sets of data from their origin inside the PIC-circuit and the server-sided pricing algorithm to their destinations.

5.3. I2C & THE PIC

The first functionality to look at at the client side is the I2C-module. We need to poll data from the meter, poll status data from the valve, and issue commands to the valve. Connecting to the slave address is step one:

i2c.rs

```
use anyhow::Result;
use rppal::i2c::I2c;
```

```
pub fn connect_i2c(slave_address: u16) -> Result<I2c, Error> {
    let mut i2c = I2c::with_bus(1)?;
    i2c.set_slave_address(slave_address)?;
    return Ok(i2c);
}
```

A simple function to return a **I2C**-Struct. Next, let's poll the meter's data from the PIC. As can be seen in the following code block, we first handle any error occurring while trying to connect to the I2C-address, then set a command that will trigger the PIC to send consumption data, send the command while propagating corresponding errors, and finally await the set of data to store into a buffer. Lastly, we return the buffer, while again handling errors that might occur. The style of error handling will be omitted where similar to keep the space in this work neat and tidy.
As you might notice, the buffer is filled with four 8-bit unsigned integers. The data is encoded in binary-coded decimals (bcd), meaning that we can hold 8 values of 4 bit size each. We will come to decoding these later.

i2c.rs

```
use crate::error::{Error, ErrorKind};

pub fn request_data(addr: u16) -> Result<[u8; 4], Error> {
    let mut i2c = match connect_i2c(addr) {                    // Create I2C Struct
        Ok(i2c) => i2c,
        Err(e) => {                                            // As seen in the connect_i2c()-
            return Err(Error {                                 // function, the only Errors are a
                kind: ErrorKind::PickNoConnection,             // error from inside the rppal-crate,
                msg: e.to_string(),                            // hinting at issues with the hard-
            })                                                 // ware interface, and an error while
        }                                                      // setting the slave_address.
    };
    let cmd = &[0x47_u8, 0x43_u8];                             // The polling command, "GC"
    let mut read = [0; 4];                                     // An empty buffer
    if let Err(e) = i2c.write(cmd) {                           // Writing command to i2c
        return Err(Error {
            kind: ErrorKind::PickWriteError,
            msg: e.to_string(),
        });
    }
```

```
    if let Err(e) = i2c.read(&mut read) {                    // Filling the buffer
        return Err(Error {
            kind: ErrorKind::PickReadError,
            msg: e.to_string(),
        });
    }
    return Ok(read);
}
```

Status requests are done similarly. As can be seen in the code blocks, the sets of data are stored as **u8**'s in arrays of fixed length, 3 bits and 4 bits, respectively. Figure XVII gives an overview over the syntax, and figure XVIII shows how to decode the status message polled from the PIC-circuit.

i2c.rs

```
pub fn request_status(addr: u16) -> Result<[u8; 3], Error> {
    let mut i2c = match connect_i2c(addr) {                  // Create I2C Struct
        Ok(i2c) => i2c,
        Err(e) => {return Err( --- snip --- )}
    };
    let cmd = &[0x47_u8, 0x56_u8];                           // Polling Command, "GV"
    let mut read = [0; 3];                                   // Empty Buffer
    if let Err(e) = i2c.write(cmd) {                         // Writing to i2c
        return Err( --- snip --- );
    }
    match i2c.read(&mut read) {                              // Filling the buffer
        Err(e) => {return Err( --- snip --- )}
        Ok(_) => {
            return Ok(read);
        }
    }
}
```

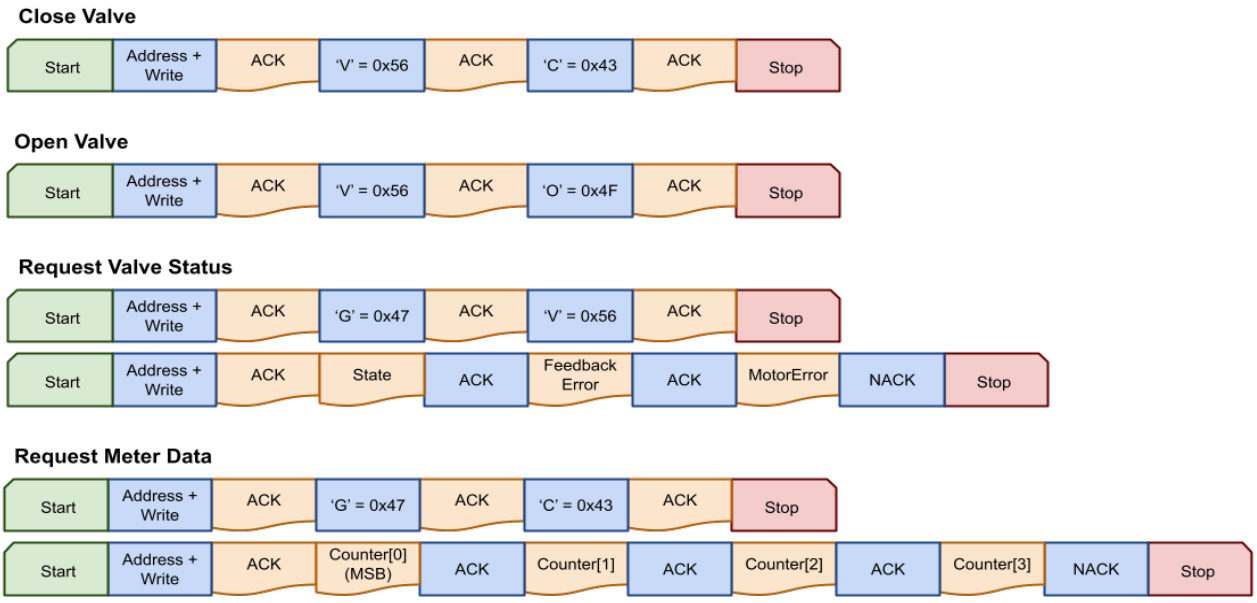

Figure XVII: Visualisation of the I2C-sequences as designed by Dipl. Ing. C. Flegel.

u8 value	**Corresponding Error / Status**
State	
0	Undefined
1	Opening
2	Opened
3	Closing
4	Closed
5	Error
FeedbackError	
0	Ok
1	Error
MotorError	
0	Ok
1	NotWorking
2	OverCurrent
3	MotorBlocked

Figure XVIII: Decoding the valve status message array.

5.4. TANGLE-RELATED FUNCTIONALITY

Now that we have a set of data available, we need to create a IOTA Streams channel to send over. Creating a channel requires us to exchange announcement and keyload messages via some conventional way. As explained in *Chapter 2.3. Scalability*, we aim to transfer this sensitive data over technician software, making logging and commissioning easier. In this work, we transfer data directly to the server via TCP/IP sockets. The **std::net** library of Rust provides all necessary structs and methods to connect to a server:

tcp.rs

```
use std::{
    net::TcpStream,
    thread
};

let mut stream = loop {
    if let Ok(s) = TcpStream::connect(addr) {                    // Trying to connect to the provided address
        break s                                                  // in 100 ms increments
    } else {
        thread::sleep(Duration::from_millis(100));
    }
};
```

Server-sided, the functionality can be mirrored by using the same library:

tcp.rs

```
use std::net::TcpListener;

let listener = TcpListener::bind("0.0.0.0:7000")?;
let (mut stream, _) = listener.accept()?;
```

The typical way of creating a channel can be done using the **UserBuilder** and appending data with the related methods:

streams.rs

```
use streams::{
    id::Ed25519,
    transport::utangle::Client,
    User
};

let mut author = User::builder()
    .with_identity(Ed25519::from_seed(author_seed))          // Seed is a pseudo-random alphanumerical
    .with_psk(server_psk.to_pskid(), server_psk)             // string
    .with_transport(Client::new(node))                       // Node is the entrance node to the tangle
    .build();                                                // network
```

A subscriber is created similarly. Pre-shared keys, as well as announcement message and keyload are exchanged with the respective instances from the server, which finishes channel creation.
In case a channel already exists, while the program starts, i.e. when a power shortage shuts down the client, we can also recover the state with an in-built **User::recover()** method and synchronise the recovered users with a **User::sync()** method. This requires us to **User::backup()** first, which we do after channel creation.

With **User::send_signet_packet()** and **User::fetch_next_msgs()**, we can start communicating over the channels that we have established.

More detailed information can be found in *Appendix A: Full Program Code*.

MESSAGE STRUCTURE

In order to achieve a reasonable data size, we encode our data into a arrray of **u8**'s. This is purely for reasons of data logistics and making the message as short as possible.

Figure XIX: A message sent to the channel always consists of one control bit, a message kind indicator, one byte of placeholder space for future development and then a minimum of one byte of data.

ControlBit should be one, The **MessageKind** is either 0 - meter data, 1 - valve status, or 2 - a software error. Please note that the valve status may contain errors forwarded from the PIC-circuit, as described earlier.

5.5. THREADING & PARALLELISM

Changing a program from being single-threaded to allow parallel task execution is uncomplicated as long as no value is accessed mutably by multiple threads simultaneously. This is simply done by moving the value inside the thread's scope and therefore transfer ownership of the value to the thread.
The Tokio crate, used by the IOTA Streams library anyway, provides improved thread and task management:

main.rs

```
use tokio;

let _status_requester = tokio::spawn(async move{                    // The move keyword tells the compiler to
});                                                                 // transfer ownership of all needed variables
```

Since we want to split all main functionality into different threads, we will need to transfer the data captured by a data requester thread to the thread responsible for sending the data to the tangle.
Tokio has us covered with messaging streams, namely broadcast- and multi-publisher-single-consumer-(MPSC) channels[41]. We can copy mpsc transmitter handles into each thread and pass the receiving end into the tangle-facing thread, funnelling all gathered data into it to be sent to the server.

main.rs

```
use tokio::sync::mpsc;

let (tx, mut rx) = mpsc::channel(32);
let _status_requester = tokio::spawn({
    let tx = tx.clone();
    let mut start_rx = start_tx.subscribe();                        // Holds back thread starting until a channel
    async move {                                                    // is created
        // handle status requesting and data forwarding here
    }
});
```

By using **loop {}** we make sure that each thread starts over as soon as it has delivered its data to the tangle-facing thread. This brings a problem. Requesting status and data from the PIC-circuit is done in a matter of milliseconds, while sending a message into the tangle takes time in the order of low single-digit seconds.

This leads to the mpsc channel queue backing up with messages, that need processing. We currently work around this by limiting the requests to one every 30 seconds, but our goal is to have close-to-near-time

control, or to put it in numbers, less than 5 seconds of delay. There are many ways of fixing, for example a local error collector, that takes values from the queue, only sends the latest meter data, but sends all the collected errors. That would involve some local calculations and checks on the meter data to prevent data manipulation, or including a timestamp.
Another attack point is the time it takes to send the message to the tangle. Proof of Work can delay processing, as well as the entry node simply being overloaded. As detailed in *Chapter 2.5: Business Insights*, we plan to incorporate a node module to be able to provide the possibly needed quality of near-real-time data transfer.

5.6. WALLET UTILITY

As described in *Chapter 1.3: Leveraged Interfaces*, the wallet library uses virtual constructors to create an **AccountManager** and an **Account**, which an address can be derived from. In order to target a payment towards the server's address, we need to pass it at least once. Since a network address is not something directly secret, we opt to include the transfer in the already existing TCP/IP- exchange, making no additional effort. Important side note here is that a deeper look into security should be taken, when the wallet library is used. We opted for a working solution that has little to no security standards implemented, which should be revisited.

wallet.rs

```
use iota_wallet::{
    account::AccountHandle, account_manager::AccountManager,
    iota_client::constants::IOTA_COIN_TYPE, ClientOptions, Result,
};

let account_manager = AccountManager::builder()
    .with_client_options(client_options)
    .with_coin_type(IOTA_COIN_TYPE)
    .finish()
    .await?;
account_manager.set_stronghold_password(stronghold_password).await?;
account_manager.store_mnemonic(mnemonic).await?;

let server_account = account_manager
    .create_account()
    .with_alias("KASKADE_SERVER_ACCOUNT".to_string())
    .finish()
    .await?;
```

In order to send funds to the server, we set the output amount and define a target address, in our case, the server's wallet address. the account takes care of sending.
Interesting for developent here is, tht we can order funds from the IOTA cafe via a simple **request_funds_from_faucet()** function.

```
main.rs

request_funds_from_faucet(
    "https://faucet.chrysalis-devnet.iota.cafe",
    &addr.address().to_bech32(),
)
.await.unwrap();

let server_wallet_addr = tcp::exchange_wallet_addr(ip_addr);

--- snip ---

let outputs = vec![AddressWithAmount {
address: server_wallet_addr.to_string(),
amount: 10_000_000,
}];
let transaction = account.send_amount(outputs, None).await?;
```

5.7. COMMAND LINE INTERFACE

In order to demonstrate a working command forwarding via the tangle, we have implemented a simple command line interface to both the server and client. On the client side, an additional thread is opened to run a command line capture:

```
main.rs

use std::io;

let mut input = String::new();
io::stdin().read_line(&mut input).unwrap();
```

We can simply **match {}** on that **input** variable to execute respective code:

main.rs (server-sided)

```
match input.as_str().trim() {
    /* case 1 */ => // code,
    /* case 2 */ => // different code,
    _ => // catch-all
}
```

5.8. Server-Sided Application

This chapter goes into details about the server-sided software, its current status and prospects. We try to focus on what's different to the Client side and emphasize the differences between the two code structures. In quick comparison, we have a mirroring behaviour in regards to the TCP/IP-functionality, the streams transmitting and receiving functionality, and the associated formatting, like error types and message sequencing. The threading is built similar in a way to be easy to read and we mirrored the wallet functionality, as well as the command line interface, though naturally it has a different purpose and logic. Additional functionality that hasn't been described would be the data bench, maybe interesting for future considerations. Again, for more in-depth knowledge, we recommend to refer to *Appendix A: Full Program Code*.

Data Bench

In order to let enough room for the development of the centre points of this work, we found that a simple data bench solution suffices. The **HashMap** type in Rust has us covered. We wrap the HashMap into an **Arc<Mutex<()>>**, in order to be able to copy the handle around and ensuring data safety in an asynchronous environment.

main.rs

```
use pond::utils::{CustomerData, TargetStatus, ValveStatus};
use std::{collections::HashMap, sync::Arc};
use tokio::sync::Mutex;

let db = Arc::new(Mutex::new(HashMap::from([
    ("consumption", CustomerData::F32(0_f32)),
    ("price", CustomerData::F32(1.5_f32)),
```

```
    ("balance", CustomerData::F32(0_f32)),
    (
        "Target Status",
        CustomerData::TargetStatusMap(TargetStatus::Open),
    ),
    (
        "Valve Status",
        CustomerData::ValveStatusMap(ValveStatus::Undefined),
    ),
])));
```

Whenever there is data to read or write to the data bench, we can acquire the **Mutex**' **LockGuard**, allowing the code to pass around reading and writing permissions, as in this example from the server side:

main.rs

```
let map_write = &mut db.lock().await;
map_write.insert("Target Status", CustomerData::TargetStatusMap(TargetStatus::Closed));
```

APPENDIX A: FULL PROGRAM CODE

This Chapter depicts the full code developed in this work. It is available digitally on the appended CD and online on request via GitHub.

A.1. CLIENT-SIDED CODE

MAIN.RS

```
use iota_wallet::{
    iota_client::request_funds_from_faucet,
    AddressWithAmount,
};
use kaskade::{
    i2c::{self, ValveCmd},
    streams,
    threads::ThreadCmd,
    tcp, utils, wallet,
};
use std::io;
use tokio::{
    sync::{broadcast, mpsc},
    task::LocalSet,
    time::{sleep, Duration},
};

#[tokio::main]
async fn main() {
    // This sections works as a temporary config.
    let author_path = "./src/backups/author.backup";
    let subscriber_path = "./src/backups/subscriber.backup";
    let wallet_path = "./src/backups/wallet.stronghold";
    let node = "https://chrysalis-nodes.iota.org";
    let i2c_address = 0x55_u16;
    let ip_addr = "192.168.2.134:7000";

    let (tx, mut rx) = mpsc::channel(8192);
    let (data_tx, mut data_rx) = mpsc::channel(8192);
    let (cmd_tx, mut cmd_rx) = mpsc::channel(8192);
    let (start_tx, _start_rx) = broadcast::channel(8192);
    let _start_rx = start_tx.subscribe();

    let local = LocalSet::new();
    println!("===== Program Started =====\n");
    let _data_manager = local.spawn_local({
        let data_tx = data_tx.clone();
        let start_tx = start_tx.clone();
        async move {
            println!("TaskManager Started");
            let (mut author, mut subscriber) =
                match streams::recover_channel(author_path, subscriber_path, node).await {
                    Ok(result) => {
                        println!("Recovering Channel...");
                        result
                    }
                    Err(_) => {
                        println!("Creating Channel...");
                        streams::create_channel(author_path, subscriber_path, node, ip_addr)
```

```
                .await
                .unwrap()
            }
        };
    println!("Channel ready.");
    start_tx.send("start").unwrap();
    loop {
        while let Some(cmd) = rx.recv().await {
            match cmd {
                ThreadCmd::SendData { data } => {
                    //println!("Sending consumption data to server.");
                    let vec = utils::prepare_data(data);
                    streams::send_msg(&mut author, vec).await.unwrap();
                    println!("Current Consumption: {} m^3", utils::bcd_to_f32(data));
                }
                ThreadCmd::SendError { err } => {
                    println!("Sending Error to server: [{}]", err);
                    let vec = utils::prepare_err(err);
                    streams::send_msg(&mut author, vec).await.unwrap();
                }
                ThreadCmd::SendStatus { status } => {
                    println!("Sending Status and PickErrors to server: [{:?}]", status);
                    let (status, error1, error2) = utils::split_status(status);
                    if error1 != 0 {
                        let vec = utils::prepare_err(utils::meter_err_to_error(error1));
                        streams::send_msg(&mut author, vec).await.unwrap();
                    }
                    if error2 != 0 {
                        let vec = utils::prepare_err(utils::motor_err_to_error(error2));
                        streams::send_msg(&mut author, vec).await.unwrap();
                    }
                    let vec = utils::prepare_status(status);
                    streams::send_msg(&mut author, vec).await.unwrap();
                }
                ThreadCmd::FetchData => {
                    println!("Listening on data from the server...");
                    let (_, data) = streams::fetch_msgs(&mut subscriber).await.unwrap();
                    data_tx.send(data).await.unwrap();
                }
            }
        }
    }
});

let _status_requester = tokio::spawn({
    let tx = tx.clone();
    let mut start_rx = start_tx.subscribe();
    async move {
        println!("StatusRequester Started");
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        loop {
            match i2c::request_status(i2c_address) {
                Err(e) => {
                    let cmd = ThreadCmd::SendError { err: e };
                    tx.send(cmd).await.unwrap();
                }
                Ok(valve_status) => {
                    let cmd = ThreadCmd::SendStatus {
                        status: valve_status,
                    };
```

```
                tx.send(cmd).await.unwrap();
            }
        }
        sleep(Duration::from_secs(30)).await;
    }
  }
});

let _data_requester = tokio::spawn({
  let tx = tx.clone();
  let mut start_rx = start_tx.subscribe();
  async move {
    println!("DataRequester Started");
    while start_rx.recv().await != Ok("start") {
      sleep(Duration::from_millis(100)).await;
    }
    loop {
      match i2c::request_data(i2c_address) {
        Err(e) => {
          let cmd = ThreadCmd::SendError { err: e };
          tx.send(cmd).await.unwrap();
        }
        Ok(data) => {
          let cmd = ThreadCmd::SendData { data: data };
          tx.send(cmd).await.unwrap();
        }
      }
      sleep(Duration::from_secs(30)).await;
    }
  }
});

let _command_fetcher = tokio::spawn({
  let tx = tx.clone();
  let mut start_rx = start_tx.subscribe();
  async move {
    println!("CommandFetcher Started");
    while start_rx.recv().await != Ok("start") {
      sleep(Duration::from_millis(100)).await;
    }
    loop {
      let cmd = ThreadCmd::FetchData;
      tx.send(cmd).await.unwrap();
      // TODO: replace data_tx/data_rx with oneshot channel and include the channel in the cmd
      while let Ok(result) = data_rx.try_recv() {
        for element in result {
          let cmd = element.0;
          let price = element.1;
          let consumption = element.2;
          cmd_tx.send(cmd).await.unwrap();
          println!(
            "Current Consumption: {} Current Price: {} € / m^3",
            consumption, price
          );
        }
      }
      sleep(Duration::from_secs(30)).await;
    }
  }
});

let _valve_controller = tokio::spawn({
  let mut start_rx = start_tx.subscribe();
```

```
    async move {
        println!("ValveControl Started");
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        loop {
            while let Ok(thread_cmd) = cmd_rx.try_recv() {
                match thread_cmd {
                    ValveCmd::Close => {
                        println!("Command retrieved: Closing Valve!");
                        i2c::close_valve(i2c_address).unwrap();
                    }
                    ValveCmd::Open => {
                        println!("Opening Valve");
                        i2c::open_valve(i2c_address).unwrap();
                    }
                }
            }
        }
    }
});

let _cli = tokio::spawn({
    let mut start_rx = start_tx.subscribe();
    async move {
        println!("WalletService started");
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        let (_manager, handle) = match wallet::recover_wallet_manager().await {
            Ok(manager) => {
                let mut accounts = manager.recover_accounts(0, 0, 2, None).await.unwrap();
                let account = accounts.pop().unwrap();
                let _balance = account.sync(None).await.unwrap();
                (manager, account)
            }
            Err(_) => {
                let (manager, account) =
                    wallet::create_wallet_manager(wallet_path).await.unwrap();
                (manager, account)
            }
        };
        let addresses = handle.generate_addresses(1, None).await.unwrap();
        let addr = match addresses.get(0) {
            Some(addr) => addr,
            None => panic!("No addr generated"),
        };

        request_funds_from_faucet(
            "https://faucet.chrysalis-devnet.iota.cafe",
            &addr.address().to_bech32(),
        )
        .await.unwrap();
        let bech = &addr.address().to_bech32();
        let server_wallet_addr = tcp::exchange_wallet_addr(bech.to_string(), ip_addr).unwrap();

        println!(
            "Your Balance: {}",
            handle.balance().await.unwrap().base_coin.available
        );
        loop {
            let mut input = String::new();
            io::stdin().read_line(&mut input).unwrap();
```

```
            match input.as_str().trim() {
                "pay MIOTA 10" => {
                    let outputs = vec![AddressWithAmount {
                    address: server_wallet_addr.to_string(),
                    amount: 10_000_000,
                    }];
                    let _transaction = handle.send_amount(outputs, None).await.unwrap();
                }
                _ => println!("Didn't recognize command, please retry."),
            }
            println!("New Balance: {}", handle.balance().await.unwrap().base_coin.available);
        }
    }
  });
  local.await;
}
```

LIB.RS

```
pub mod error;
pub mod i2c;
pub mod streams;
pub mod tcp;
pub mod threads;
pub mod utils;
pub mod wallet;
```

STREAMS.RS

```
use crate::{
  i2c::ValveCmd,
  tcp::{exchange_links, exchange_psks},
  utils::{self, gen_seed},
};
use anyhow::{self, bail};
use std::{
  fs::{self, File},
  io::Write,
  str,
};
use streams::{
  id::{Ed25519, Psk},
  transport::utangle::Client,
  Address, User,
};

const TOPIC: &str = "KASKADENET";

/// Recovers User data from a backup file in case of unexpected shutdowns before returning both Users.
pub async fn recover_channel(
  author_path: &str,
  subscriber_path: &str,
  node: &str,
) -> anyhow::Result<(User<Client>, User<Client>)> {

  // TODO: Disable PoW.
  let author_backup = fs::read(author_path)?;
  let mut author = User::restore(author_backup, "", Client::new(node)).await?;
  author.sync().await?;

  let subscriber_backup = fs::read(subscriber_path)?;
  let mut subscriber = User::restore(subscriber_backup, "", Client::new(node)).await?;
  subscriber.sync().await?;
```

```
    return Ok((author, subscriber));
}

/// This function creates two users from scratch. The Author is connected to the server's subscriber, and vice versa.
/// PSK transfer is done via TCP, as well as publishing of the keyload. Calls to store User data before returning both
Users.
pub async fn create_channel(
    author_path: &str,
    subscriber_path: &str,
    node: &str,
    ip_addr: &str,
) -> anyhow::Result<(User<Client>, User<Client>)> {

    // TODO: Disable PoW.
    let author_seed = gen_seed();
    let subscriber_seed = gen_seed();
    let client_psk = Psk::from_seed(gen_seed());

    let mut author = User::builder()
        .with_identity(Ed25519::from_seed(author_seed))
        .with_psk(client_psk.to_pskid(), client_psk)
        .with_transport(Client::new(node))
        .build();
    let server_psk = exchange_psks(ip_addr, client_psk)?;
    let mut subscriber = User::builder()
        .with_identity(Ed25519::from_seed(subscriber_seed))
        .with_psk(server_psk.to_pskid(), server_psk)
        .with_transport(Client::new(node))
        .build();

    let announcement = author.create_stream(TOPIC).await?;
    let keyload = author.send_keyload_for_all(TOPIC).await?;

    let server_ann_link = exchange_links(ip_addr, announcement.address())?;
    let server_keyload = exchange_links(ip_addr, keyload.address())?;
    subscriber.receive_message(server_ann_link).await?;
    subscriber.receive_message(server_keyload).await?;

    store_rec_data(author_path, &mut author).await?;
    store_rec_data(subscriber_path, &mut subscriber).await?;

    return Ok((author, subscriber));
}

/// Stores streams User data to backup file.
async fn store_rec_data(path: &str, user: &mut User<Client>) -> anyhow::Result<()> {
    let backup = user.backup("").await?;
    println!("storing backup to file: {}", path);
    let mut file = File::create(path)?;
    file.write(&backup)?;
    Ok(())
}

/// Formats and sends message to tangle and returns message link.
pub async fn send_msg(
    user: &mut User<Client>,
    vec: Vec<u8>,
) -> anyhow::Result<(&mut User<Client>, Address)> {
    let msg = String::from_utf8(vec)?;
    let packet = user.send_signed_packet(TOPIC, &msg, &msg).await?;
    return Ok((user, packet.address()));
}
```

```
/// Returns a vector over all new messages fetched from the tangle.
pub async fn fetch_msgs(
    user: &mut User<Client>,
) -> anyhow::Result<(&mut User<Client>, Vec<(ValveCmd, f32, f32)>)> {
    let messages = user.fetch_next_messages().await?;
    let mut results: Vec<(ValveCmd, f32, f32)> = vec![];
    for msg in messages {
        if msg.is_signed_packet() {
            let content = match msg.as_signed_packet() {
                Some(content) => content,
                None => bail!("Message is not a signed packet!"),
            };
            let result = (
                utils::refactor_valve_cmd(content.masked_payload[1])?,
                f32::from_be_bytes(content.masked_payload[2..=5].try_into()?),
                f32::from_be_bytes(content.masked_payload[6..=9].try_into()?),
            );
            results.push(result);
        }
    }
    Ok((user, results))
}
```

WALLET.RS

```
use std::path::PathBuf;
use iota_wallet::{
    account::AccountHandle,
    account_manager::AccountManager,
    iota_client::{constants::IOTA_COIN_TYPE, utils::generate_mnemonic},
    secret::{mnemonic::MnemonicSecretManager, stronghold::StrongholdSecretManager, SecretManager},
    ClientOptions, Result,
};

pub async fn create_wallet_manager(backup_path: &str) -> Result<(AccountManager, AccountHandle)>
    let stronghold_password = "KASKADE_SECURE_PASSWORD";
    let mnemonic = generate_mnemonic()?;
    let mnemonic = mnemonic.as_str();
    let node = "https://api.lb-1.h.chrysalis-devnet.iota.cafe";

    let mut secret_manager = StrongholdSecretManager::builder()
        .password(stronghold_password)
        .build(PathBuf::from("backup_path"))?;
    secret_manager.store_mnemonic(mnemonic.to_string()).await?;
    let client_options = ClientOptions::new().with_node(node)?;
    let account_manager = AccountManager::builder()
        .with_secret_manager(SecretManager::Stronghold(secret_manager))
        .with_client_options(client_options)
        .with_coin_type(IOTA_COIN_TYPE)
        .finish()
        .await?;
    account_manager
        .set_stronghold_password(stronghold_password)
        .await?;
    account_manager.store_mnemonic(mnemonic.to_string()).await?;

    let client_account = account_manager
        .create_account()
        .with_alias("KASKADE_CLIENT_ACCOUNT".to_string())
        .finish()
        .await?;
    //println!("Generated a new account: {:?}", client_account);
    account_manager
```

```
        .backup(PathBuf::from(backup_path), stronghold_password.to_string())
        .await?;

    Ok((account_manager, client_account))
}

pub async fn recover_wallet_manager() -> Result<AccountManager> {
    let mnemonic = generate_mnemonic()?;
    let mnemonic = mnemonic.as_str();
    let node = "https://api.lb-0.h.chrysalis-devnet.iota.cafe";
    let client_options = ClientOptions::new().with_node(node)?;
    let secret_manager = MnemonicSecretManager::try_from_mnemonic(mnemonic)?;
    let manager = AccountManager::builder()
        .with_secret_manager(SecretManager::Mnemonic(secret_manager))
        .with_client_options(client_options)
        .with_coin_type(IOTA_COIN_TYPE)
        .finish()
        .await?;

    Ok(manager)
}
```

TCP.RS

```
use std::{
    io::{Read, Write},
    net::TcpStream,
    str::{self, FromStr},
    thread,
    time::Duration,
};
use streams::{id::Psk, Address};

/// Opens TCP-connection, sends a link and then awaits the server's link.
pub fn exchange_links(addr: &str, link: Address) -> anyhow::Result<Address> {
    println!("Exchanging Addresses...");
    let mut stream = loop {
        if let Ok(s) = TcpStream::connect(addr) {
            break s
        } else {
            thread::sleep(Duration::from_millis(100));
        }
      };

    stream.write(&link.to_string().as_bytes()).unwrap();

    let mut incoming = [0_u8; 200];
    let size = stream.read(&mut incoming).unwrap();
    let address = str::from_utf8(&incoming[0..size]).unwrap();

    Ok(Address::from_str(address).unwrap())
}

/// Opens TCP-connection, sends a Psk and then awaits the server's Psk.
pub fn exchange_psks(addr: &str, psk: Psk) -> anyhow::Result<Psk> {
    println!("Exchanging Pre-Shared Keys...");
    let mut stream = loop {
        if let Ok(s) = TcpStream::connect(addr) {
            break s
        } else {
            thread::sleep(Duration::from_millis(100));
        }
      };
```

```
    stream.write(&psk.as_ref()).unwrap();

    let mut msg = [0_u8; 32];
    stream.read(&mut msg).unwrap();

    Ok(Psk::new(msg))
}

pub fn exchange_wallet_addr(wall_addr: String, addr: &str) -> anyhow::Result<String> {
    println!("Exchanging wallet addresses...");
    let mut stream = loop{
        if let Ok(s) = TcpStream::connect(addr) {
            break s
        } else {
            thread::sleep(Duration::from_millis(100));
        }
    };

    stream.write(&wall_addr.to_string().as_bytes())?;

    let mut msg = [0_u8; 200];
    let size = stream.read(&mut msg)?;
    let address = str::from_utf8(&msg[0..size])?;

    Ok(String::from(address))
}
```

I2C.RS

```
use crate::error::{Error, ErrorKind};
use anyhow::Result;
use rppal::i2c::I2c;

/// Polls data from sensor. Data is stored in an Array, return Ok(read);.
/// # Syntax
/// The content of the Array isn't read on the client side, only forwarded to the server.
/// read is a bcd, 1 byte is split into 2x4 bits, each group represents an f32.
pub fn request_data(addr: u16) -> Result<[u8; 4], Error> {
    let mut i2c = match connect_i2c(addr) {
        Ok(i2c) => i2c,
        Err(e) => {
            return Err(Error {
                kind: ErrorKind::PickNoConnection,
                msg: e.to_string(),
            })
        }
    };
    let cmd = &[0x47_u8, 0x43_u8];
    let mut read = [0; 4];
    if let Err(e) = i2c.write(cmd) {
        return Err(Error {
            kind: ErrorKind::PickWriteError,
            msg: e.to_string(),
        });
    }
    if let Err(e) = i2c.read(&mut read) {
        return Err(Error {
            kind: ErrorKind::PickReadError,
            msg: e.to_string(),
        });
    }
    return Ok(read);
}
```

```
/// Polls the pick's status values. Returns all three values.
/// Value 1 -> State.
///     0 => Undefined.
///     1 => Opening.
///     2 => Opened.
///     3 => Closing.
///     4 => Closed.
///     5 => Error.
/// Value 2 -> FeedBackError. Stored by the pick on error with the water meter.
///     0 => OK.
///     1 => Error.
/// Value 3 -> MotorError. Stored on error with the valve.
///     0 => OK.
///     1 => NotWorking.
///     2 => OverCurrent.
///     3 => MotorBlocked.
pub fn request_status(addr: u16) -> Result<[u8; 3], Error> {
    let mut i2c = match connect_i2c(addr) {
        Ok(i2c) => i2c,
        Err(e) => {
            return Err(Error {
                kind: ErrorKind::PickNoConnection,
                msg: e.to_string(),
            })
        }
    };
    let cmd = &[0x47_u8, 0x56_u8];
    let mut read = [0; 3];
    if let Err(e) = i2c.write(cmd) {
        return Err(Error {
            kind: ErrorKind::PickWriteError,
            msg: e.to_string(),
        });
    }
    match i2c.read(&mut read) {
        Err(e) => {
            return Err(Error {
                kind: ErrorKind::PickReadError,
                msg: e.to_string(),
            })
        }
        Ok(_) => {
            return Ok(read);
        }
    }
}

/// Closes the valve.
pub fn close_valve(addr: u16) -> Result<(), Error> {
    let mut i2c = match connect_i2c(addr) {
        Ok(i2c) => i2c,
        Err(e) => {
            return Err(Error {
                kind: ErrorKind::PickNoConnection,
                msg: e.to_string(),
            })
        }
    };
    let cmd = &[0x56_u8, 0x43_u8];
    if let Err(e) = i2c.write(cmd) {
        return Err(Error {
            kind: ErrorKind::PickWriteError,
```

```
            msg: e.to_string(),
        });
    }
    return Ok(());
}

/// Opens the valve.
pub fn open_valve(addr: u16) -> Result<(), Error> {
    let mut i2c = match connect_i2c(addr) {
        Ok(i2c) => i2c,
        Err(e) => {
            return Err(Error {
                kind: ErrorKind::PickNoConnection,
                msg: e.to_string(),
            })
        }
    };
    let cmd = &[0x56_u8, 0x4F_u8];
    if let Err(e) = i2c.write(cmd) {
        return Err(Error {
            kind: ErrorKind::PickWriteError,
            msg: e.to_string(),
        });
    }
    return Ok(());
}

pub fn connect_i2c(slave_address: u16) -> Result<I2c, Error> {
    let mut i2c = I2c::with_bus(1).unwrap();
    i2c.set_slave_address(slave_address)?;
    return Ok(i2c);
}

#[derive(Debug)]
pub enum ValveStatus {
    Initializing,
    Opening,
    Open,
    Closing,
    Closed,
}
#[derive(Clone, Copy, Debug)]
pub enum ValveCmd {
    Open,
    Close,
}
```

UTILS.RS

```
use crate::{
    error::{Error, ErrorKind},
    i2c::ValveCmd,
};
use anyhow::{self, bail};
use rand::{distributions::Alphanumeric, thread_rng, Rng};

/// Generates a 32-spaces alphanumerical pseudo-random string
pub fn gen_seed() -> String {
    thread_rng()
        .sample_iter(&Alphanumeric)
        .take(32)
        .map(char::from)
        .collect()
```

```
}

// formats data from a [u8; 4] into the sent vector.
pub fn prepare_data(s: [u8; 4]) -> Vec<u8> {
    let mut res = vec![1, 0, 0];
    for i in s {
        res.push(i);
    }
    res
}

pub fn prepare_err(err: Error) -> Vec<u8> {
    let mut res = vec![1, 2, 0];
    res.push(err.kind as u8);
    res
}

pub fn prepare_status(status: u8) -> Vec<u8> {
    let mut res = vec![1, 1, 0];
    res.push(status);
    res
}

pub fn split_status(status: [u8; 3]) -> (u8, u8, u8) {
    (status[0], status[1], status[2])
}

pub fn meter_err_to_error(error: u8) -> Error {
    let err = match error {
        1 => Error {
            kind: ErrorKind::ValveFeedBack,
            msg: "Water meter returned no feedback.".to_string(),
        },
        _ => Error {
            kind: ErrorKind::OutOfRange,
            msg: "Meter error bit exceeded 1, but the highest value is 1.".to_string(),
        },
    };
    err
}

pub fn motor_err_to_error(error: u8) -> Error {
    let err = match error {
        1 => Error {
            kind: ErrorKind::MotorNotWorking,
            msg: "Valve motor out of order.".to_string(), // Better Description needed.
        },
        2 => Error {
            kind: ErrorKind::MotorOverCurrent,
            msg: "Valve motor has overcurrent.".to_string(), // Better Description needed.
        },
        3 => Error {
            kind: ErrorKind::MotorBlocked,
            msg: "Valve is physically blocked.".to_string(),
        },
        _ => Error {
            kind: ErrorKind::OutOfRange,
            msg: "Motor error bit exceeded 3, but the highest value is 3.".to_string(),
        },
    };
    err
}
```

```
pub fn refactor_valve_cmd(valve_cmd: u8) -> anyhow::Result<ValveCmd> {
    let res = match valve_cmd {
        0 => Ok(ValveCmd::Open),
        1 => Ok(ValveCmd::Close),
        _ => bail!("OutOfRange"),
        };
    res
}

pub fn bcd_to_f32(data: [u8; 4]) -> f32 {
    //println!("bcd_debug: data: {:?}", data);
    let mut vec: Vec<u8> = vec![];
    for elem in data {
        let elem_high = (elem & 0xf0)/16;
        let elem_low = elem & 0x0f;
        //println!("bcd_debug: elem: {}, elem_high: {}, elem_low: {}", elem, elem_high, elem_low);
        vec.push(elem_high);
        vec.push(elem_low);
    }
    let mut res: Vec<f32> = vec![];
    let mut j = 10000_f32;
    for i in vec {
        let mut i: f32 = i.into();
        i *= j;
        res.push(i);
        j /= 10_f32;
    }
    res.iter().sum()
}

pub fn refactor_as_f32(a: [u8; 4]) -> anyhow::Result<f32> {
    Ok(f32::from_be_bytes(a))
}
```

ERROR.RS

```
use std::fmt;
use thiserror::Error;

#[derive(Clone, Debug)]
pub struct Error {
    pub kind: ErrorKind,
    pub msg: String,
}

#[derive(Clone, Debug, Error)]
pub enum ErrorKind {
    #[error("")]
    ValveNoResponse,
    #[error("")]
    ValveMismatchedState,
    #[error("")]
    ValveUnknownStatus,
    #[error("")]
    MeterNoResponse,
    #[error("")]
    MeterNoValidData,
    #[error("")]
    MotorBlocked,
    #[error("")]
    MotorNotWorking,
    #[error("")]
    MotorOverCurrent,
```

```
    #[error("")]
    PickError,
    #[error("")]
    ValveFeedBack,
    #[error("")]
    PickNoConnection,
    #[error("")]
    PickReadError,
    #[error("")]
    PickWriteError,
    #[error("OutOfRange")]
    OutOfRange,
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(
            f,
            "An Error occured: {:#?} with msg: {}",
            self.kind, self.msg
        )
    }
}

impl From<rppal::i2c::Error> for Error {
    fn from(error: rppal::i2c::Error) -> Self {
        Error {
            kind: ErrorKind::PickError,
            msg: error.to_string(),
        }
    }
}
```

THREADS.RS

```
use crate::error::Error;

#[derive(Debug)]
pub enum ThreadCmd {
    SendData {
        data: [u8; 4],
    },
    SendStatus {
        status: [u8; 3],
    },
    SendError {
        err: Error,
    },
    FetchData,
}
```

A.2. SERVER-SIDED CODE

MAIN.RS

```
use pond::{
    streams,
    tcp,
    threads::ThreadCmd,
    utils::{self, CustomerData, TargetStatus, ValveStatus},
    wallet,
```

```
};
use std::{
    collections::HashMap,
    io,
    sync::Arc,
};
use tokio::{
    sync::{broadcast, mpsc, Mutex},
    task::LocalSet,
    time::{sleep, Duration},
};

#[tokio::main]
async fn main() {
    // This sections works as a temporary config.
    let author_path = "./src/backups/author.backup";
    let subscriber_path = "./src/backups/subscriber.backup";
    let wallet_path = "./src/backups/wallet.stronghold";
    let node = "https://chrysalis-nodes.iota.org";

    // A HashMap, the simplest data bench there is.
    let db = Arc::new(Mutex::new(HashMap::from([
        ("consumption", CustomerData::F32(0_f32)),
        ("price", CustomerData::F32(1.5_f32)),
        ("balance", CustomerData::F32(0_f32)),
        (
            "Target Status",
            CustomerData::TargetStatusMap(TargetStatus::Open),
        ),
        (
            "Valve Status",
            CustomerData::ValveStatusMap(ValveStatus::Undefined),
        ),
    ])));

    let (tx, mut rx) = mpsc::channel(8192);
    let (start_tx, _start_rx) = broadcast::channel(8192);

    let local = LocalSet::new();
    println!("===== Program Started =====\n");

    let _data_manager = local.spawn_local({
        let start_tx = start_tx.clone();
        let db = db.clone();
        async move {
            println!("TaskManager Started");
            let (mut author, mut subscriber) =
                match streams::recover_channel(author_path, subscriber_path, node).await {
                    Ok(result) => {
                        println!("Recovering Channel...");
                        result
                    }
                    Err(_) => {
                        println!("Creating Channel...");
                        streams::create_channel(author_path, subscriber_path, node)
                            .await
                            .unwrap()
                    }
                };
            println!("Channel ready.");
            start_tx.send("start").unwrap();
            loop {
                while let Some(cmd) = rx.recv().await {
```

```
            match cmd {
                ThreadCmd::FetchData => {
                    let messages = subscriber.fetch_next_messages().await.unwrap();
                    if let Some(_) = messages.get(0) {
                        for msg in messages {
                            if msg.is_signed_packet() {
                                let inner = &msg.as_signed_packet().unwrap().masked_payload;
                                match inner[1] {
                                    0 => {
                                        let data =
                                            utils::bcd_to_f32(inner[3..=6].try_into().unwrap());
                                        let mut map = db.lock().await;
                                        println!("{}", utils::handle_data(&mut map, data));
                                    }
                                    1 => {
                                        let mut map = db.lock().await;
                                        println!(
                                            "{}",
                                            utils::handle_status(&mut map, inner[3])
                                        );
                                    }
                                    2 => {
                                        println!("{}", utils::handle_error(inner[3]));
                                    }
                                    _ => {
                                        // return OutOfRange Error.
                                    }
                                }
                            }
                            utils::algo(&mut db.lock().await);
                        }
                    }
                }
                ThreadCmd::SendData => {
                    let map_read = db.lock().await;
                    let Some(CustomerData::TargetStatusMap(cmd)) = map_read.get("Target Status") else {panic!()};
                    let Some(CustomerData::F32(cons)) = map_read.get("consumption") else {panic!()};
                    let Some(CustomerData::F32(price)) = map_read.get("price") else {panic!()};
                    let vec = utils::prepare_data(cmd, *price, *cons);
                    println!("Sent Data: {:?}", vec);
                    streams::send_msg(&mut author, vec).await.unwrap();
                }
            }
        }
    }
});

let _wallet_manager = tokio::spawn({
    let _tx = tx.clone();
    let mut start_rx = start_tx.subscribe();
    let db = db.clone();
    async move {
        println!("WalletManager started.");
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        let (_manager, handle) = match wallet::recover_wallet_manager().await {
            Ok(manager) => {
                let mut accounts = manager.recover_accounts(0, 0, 2, None).await.unwrap();
                let account = accounts.pop().unwrap();
                let _balance = account.sync(None).await.unwrap();
                (manager, account)
```

```
            }
            Err(_) => {
                let (manager, account) =
                    wallet::create_wallet_manager(wallet_path).await.unwrap();
                (manager, account)
            }
        };
        let addresses = handle.generate_addresses(1, None).await.unwrap();
        let addr = match addresses.get(0) {
            Some(addr) => addr,
            None => panic!("No addr generated"),
        };
        let bech = &addr.address().to_bech32();
        tcp::exchange_wallet_addr(bech.to_string()).unwrap();
        loop {
            let balance = handle.balance().await.unwrap();
            let map_write = &mut db.lock().await;
            map_write.insert(
                "balance",
                CustomerData::F32(balance.base_coin.available as f32),
            );
            sleep(Duration::from_secs(5)).await;
        }
    }
});

let _tangle_receiver = tokio::spawn({
    let tx = tx.clone();
    let mut start_rx = start_tx.subscribe();
    async move {
        println!("TangleReceiver started");
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        loop {
            let cmd = ThreadCmd::FetchData;
            tx.send(cmd).await.unwrap();
            sleep(Duration::from_secs(30)).await;
        }
    }
});

let _tangle_sender = tokio::spawn({
    let tx = tx.clone();
    let mut start_rx = start_tx.subscribe();
    async move {
        println!("TangleCmdSender started");
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        loop {
            let cmd = ThreadCmd::SendData;
            tx.send(cmd).await.unwrap();
            sleep(Duration::from_secs(30)).await;
        }
    }
});

let _cli = tokio::spawn({
    let mut start_rx = start_tx.subscribe();
    let db = db.clone();
    async move {
        println!("CLI started");
```

```
        while start_rx.recv().await != Ok("start") {
            sleep(Duration::from_millis(100)).await;
        }
        loop {
            let mut input = String::new();
            io::stdin().read_line(&mut input).unwrap();
            match input.as_str().trim() {
                "valve close" => {
                    let map_write = &mut db.lock().await;
                    map_write.insert("Target Status", CustomerData::TargetStatusMap(TargetStatus::Closed));
                },
                "valve open" => {
                    let map_write = &mut db.lock().await;
                    map_write.insert("Target Status", CustomerData::TargetStatusMap(TargetStatus::Open));
                },
                _ => println!("Didn't recognize command, please retry.")
            }
        }
    }
});

local.await;
}
```

LIB.RS

```
pub mod streams;
pub mod error;
pub mod threads;
pub mod tcp;
pub mod utils;
pub mod wallet;
```

STREAMS.RS

```
use crate::{
    error::Error,
    tcp,
    utils,
};
use std::{
    fs::{self, File},
    io::Write,
    str,
};
use streams::{
    id::{Ed25519, Psk},
    transport::utangle::Client,
    Address, User
};

const TOPIC: &str = "KASKADENET";

/// Recovers User data from a backup file in case of unexpected shutdowns before returning both Users.
pub async fn recover_channel(
    author_path: &str,
    subscriber_path: &str,
    node: &str,
) -> anyhow::Result<(User<Client>, User<Client>)> {

    // TODO: Disable PoW.
    let author_backup = fs::read(author_path)?;
    let mut author = User::restore(author_backup, "", Client::new(node)).await?;
```

```
    author.sync().await?;

    let subscriber_backup = fs::read(subscriber_path)?;
    let mut subscriber = User::restore(subscriber_backup, "", Client::new(node)).await?;
    subscriber.sync().await?;

    return Ok((author, subscriber));
}

/// This function creates two users from scratch. The Author is connected to the server's subscriber, and vice versa.
/// PSK transfer is done via TCP, as well as publishing of the keyload. Calls to store User data before returning both
Users.
pub async fn create_channel(
    author_path: &str,
    subscriber_path: &str,
    node: &str,
) -> anyhow::Result<(User<Client>, User<Client>)> {

    // TODO: Disable PoW.
    let author_seed = utils::gen_seed();
    let subscriber_seed = utils::gen_seed();
    let server_psk = Psk::from_seed(utils::gen_seed());

    let mut author = User::builder()
        .with_identity(Ed25519::from_seed(author_seed))
        .with_psk(server_psk.to_pskid(), server_psk)
        .with_transport(Client::new(node))
        .build();

    let client_psk = tcp::exchange_psks(server_psk).unwrap();
    let mut subscriber = User::builder()
        .with_identity(Ed25519::from_seed(subscriber_seed))
        .with_psk(client_psk.to_pskid(), client_psk)
        .with_transport(Client::new(node))
        .build();

    let announcement = author.create_stream(TOPIC).await?;
    let keyload = author.send_keyload_for_all(TOPIC).await?;

    let client_ann_link = tcp::exchange_links(announcement.address())?;
    let client_keyload = tcp::exchange_links(keyload.address())?;
    subscriber.receive_message(client_ann_link).await?;
    subscriber.receive_message(client_keyload).await?;

    store_rec_data(author_path, &mut author).await;
    store_rec_data(subscriber_path, &mut subscriber).await;

    println!("created:\nauthor: {:?}\nsubscriber: {:?}", author, subscriber);
    return Ok((author, subscriber));
}

/// Stores streams User data to backup file.
async fn store_rec_data(path: &str, user: &mut User<Client>) {
    let backup = user.backup("").await.unwrap();
    println!("storing to file: {}", path);
    let mut file = File::create(path).unwrap();
    file.write(&backup).unwrap();
}

/// Formats and sends message to tangle and returns message link.
pub async fn send_msg(
    user: &mut User<Client>,
    vec: Vec<u8>,
```

```
) -> Result<(&mut User<Client>, Address), Error> {
    let msg = String::from_utf8(vec).unwrap();
    let packet = user.send_signed_packet(TOPIC, &msg, &msg).await.unwrap();
    return Ok((user, packet.address()));
}
```

WALLET.RS

```
use std::path::PathBuf;
use iota_wallet::{
    account::AccountHandle,
    account_manager::AccountManager,
    iota_client::{constants::IOTA_COIN_TYPE, utils::generate_mnemonic},
    secret::{mnemonic::MnemonicSecretManager, stronghold::StrongholdSecretManager, SecretManager},
    ClientOptions, Result,
};

pub async fn create_wallet_manager(backup_path: &str) -> Result<(AccountManager, AccountHandle)> {
    let stronghold_password = "KASKADE_SECURE_PASSWORD";
    let mnemonic = generate_mnemonic()?;
    let mnemonic = mnemonic.as_str();
    let node = "https://api.lb-0.h.chrysalis-devnet.iota.cafe";

    let mut secret_manager = StrongholdSecretManager::builder()
        .password(stronghold_password)
        .build(PathBuf::from("backup_path"))?;
    secret_manager.store_mnemonic(mnemonic.to_string()).await?;
    let client_options = ClientOptions::new().with_node(node)?;
    let account_manager = AccountManager::builder()
        .with_secret_manager(SecretManager::Stronghold(secret_manager))
        .with_client_options(client_options)
        .with_coin_type(IOTA_COIN_TYPE)
        .finish()
        .await?;
    account_manager
        .set_stronghold_password(stronghold_password)
        .await?;
    account_manager.store_mnemonic(mnemonic.to_string()).await?;

    let server_account = account_manager
        .create_account()
        .with_alias("KASKADE_SERVER_ACCOUNT".to_string())
        .finish()
        .await?;
    account_manager
        .backup(PathBuf::from(backup_path), stronghold_password.to_string())
        .await?;

    Ok((account_manager, server_account))
}

pub async fn recover_wallet_manager() -> Result<AccountManager> {
    let mnemonic = generate_mnemonic()?;
    let mnemonic = mnemonic.as_str();
    let node = "https://api.lb-0.h.chrysalis-devnet.iota.cafe";
    let client_options = ClientOptions::new().with_node(node)?;
    let secret_manager = MnemonicSecretManager::try_from_mnemonic(mnemonic)?;
    let manager = AccountManager::builder()
        .with_secret_manager(SecretManager::Mnemonic(secret_manager))
        .with_client_options(client_options)
        .with_coin_type(IOTA_COIN_TYPE)
        .finish()
        .await?;
```

```
    Ok(manager)
}
```

TCP.RS

```
use std::{
    io::{Read, Write},
    net::TcpListener,
    str::{self, FromStr},
};
use streams::{
    Address,
    id::Psk,
};

pub fn exchange_links(link: Address) -> anyhow::Result<Address> {
    println!("Exchanging Addresses...");
    let listener = TcpListener::bind("0.0.0.0:7000")?;
    let (mut stream, _) = listener.accept()?;

    let mut incoming = [0_u8; 200];
    let size = stream.read(&mut incoming)?;
    let address = str::from_utf8(&incoming[0..size])?;

    stream.write(&link.to_string().as_bytes()).unwrap();

    Ok(Address::from_str(address).unwrap())
}

pub fn exchange_psks(psk: Psk) -> anyhow::Result<Psk> {
    println!("Exchanging Pre-Shared Keys...");
    let listener = TcpListener::bind("0.0.0.0:7000")?;
    let (mut stream, _) = listener.accept()?;

    let mut incoming = [0_u8; 32];
    let _size = stream.read(&mut incoming).unwrap();

    stream.write(&psk.as_ref()).unwrap();

    Ok(Psk::new(incoming))
}
pub fn exchange_wallet_addr(addr: String) -> anyhow::Result<String>{
    println!("Exchanging Wallets Addresses...");
    let listener = TcpListener::bind("0.0.0.0:7000")?;
    let (mut stream, _) = listener.accept()?;

    let mut incoming = [0_u8; 200];
    let size = stream.read(&mut incoming)?;
    let address = str::from_utf8(&incoming[0..size])?;

    stream.write(&addr.to_string().as_bytes()).unwrap();

    Ok(String::from(address))
}
```

UTILS.RS

```
use rand::{distributions::Alphanumeric, thread_rng, Rng};
use std::collections::HashMap;
use tokio::sync::MutexGuard;

pub fn handle_data(data_bench: &mut HashMap<&str, CustomerData>, data: f32) -> String {
```

```
    data_bench.insert("consumption", CustomerData::F32(data));
    String::from(format!("Current Consumption: {} m^3", data))
}

pub fn handle_status(data_bench: &mut HashMap<&str, CustomerData>, data: u8) -> String {
    data_bench.insert(
        "Valve Status",
        CustomerData::ValveStatusMap(ValveStatus::from_u8(data)),
    );
    match data {
        0 => {
            String::from("Valve Status is: Undefined")
        },
        1 => {
            String::from("Valve Status is: Opening")
        },
        2 => {
            String::from("Valve Status is: Opened")
        },
        3 => {
            String::from("Valve Status is: Closing")
        },
        4 => {
            String::from("Valve Status is: Closed")
        },
        5 => {
            String::from("Warning: Valve Status returned an error. If this issue persists, consider checking the hardware.")
        },
        _ => {
            String::from("Warning: An unexpected Error occurred. Please report to devs.")
        },
    }
}

pub fn handle_error(data: u8) -> String {
    match data {
        0 => {
            String::from("Warning: The Valve isn't responding. Consider checking the hardware.")
        },
        1 => {
            String::from("Warning: The Valve's State is mismatched. If this issue persists, consider checking the hardware.")
        },
        2 => {
            String::from("Warning: The Valve' State is unknown. If this issue persists, consider checking the hardware.")
        },
        3 => {
            String::from("Warning: The Meter isn't responding. Consider checking the hardware.")
        },
        4 => {
            String::from("Warning: The Meter sent invalid data. If this issue persists, consider checking the hardware.")
        },
        5 => {
            String::from("Warning: The Motor is blocked. If this issue persists, consider checking the hardware.")
        },
        6 => {
            String::from("Warning: The Motor isn't working. Consider checking the hardware.")
        },
        7 => {
            String::from("Warning: The Motor has had an over current. Check hardware immediately.")
        },
        8 => {
            String::from("Warning: The Pick has sent an unknown error.")
        },
```

```
        9 => {
            String::from("Warning: The  Valve's feedback wire is compromised. Check hardware immediately.")
        },
        10 => {
            String::from("Warning: Connection between Pick and Raspi not possible. Consider checking for Permission
settings and hardware, if this issue persists.")
        },
        11 => {
            String::from("Warning: Raspi couldn't read from i2c interface. Consider checking for Permission settings and
hardware, if this issue persists.")
        },
        12 => {
            String::from("Warning: Raspi couldn't write to i2c interface. Consider checking for Permission settings and
hardware, if this issue persists.")
        },
        13 => {
            String::from("Warning: Index out of Bound. Origin currently unknown. Data might be compromised.")
        },
        _ => {
            String::from("Warning: An unexpected Error occurred. Please report to devs.")
        },
    }
}

pub fn bcd_to_f32(data: [u8; 4]) -> f32 {
    let mut vec: Vec<u8> = vec![];
    for elem in data {
        let elem_high = (elem & 0xf0) / 16;
        let elem_low = elem & 0x0f;
        vec.push(elem_high);
        vec.push(elem_low);
    }
    let mut res: Vec<f32> = vec![];
    let mut j = 10000_f32;
    for i in vec {
        let mut i: f32 = i.into();
        i *= j;
        res.push(i);
        j /= 10_f32;
    }
    res.iter().sum()
}

#[derive(Clone, Copy)]
pub enum TargetStatus {
    Open,
    Closed,
}

pub enum ValveStatus {
    Undefined,
    Opening,
    Opened,
    Closing,
    Closed,
    Error,
}
impl ValveStatus {
    fn from_u8(a: u8) -> Self {
        match a {
            0 => Self::Undefined,
            1 => Self::Opening,
            2 => Self::Opened,
```

```
            3 => Self::Closing,
            4 => Self::Closed,
            5 => Self::Error,
            _ => Self::Undefined,
        }
    }
}

pub enum CustomerData {
    F32(f32),
    TargetStatusMap(TargetStatus),
    ValveStatusMap(ValveStatus),
}
/* impl CustomerData {
    pub fn as_f32(self) -> f32 {
        if self == CustomerData::ValveStatusMap {
            self
        }
    }
} */

pub fn algo(map_read: &mut MutexGuard<HashMap<&str, CustomerData>>) {
    let Some(CustomerData::ValveStatusMap(_status)) = map_read.get("Valve Status") else {panic!()};
    let Some(CustomerData::TargetStatusMap(_cmd)) = map_read.get("Target Status") else {panic!()};
    let Some(CustomerData::F32(cons)) = map_read.get("consumption") else {panic!()};
    let Some(CustomerData::F32(price)) = map_read.get("price") else {panic!()};

    let new_price = price + ( 0.5_f32 * cons);
    let new_status = if cons > &5_f32 {
        TargetStatus::Closed
    } else {
        TargetStatus::Open
    };
    map_read.insert("price", CustomerData::F32(new_price));
    map_read.insert("Target Status", CustomerData::TargetStatusMap(new_status));
}

pub fn prepare_data(cmd: &TargetStatus, price: f32, cons: f32, ) -> Vec<u8> {
    vec![
        *cmd as u8,
        price as u8,
        cons as u8
    ]
}

/// Generates a 32-spaces alphanumerical pseudo-random string.
pub fn gen_seed() -> String {
    thread_rng()
        .sample_iter(&Alphanumeric)
        .take(32)
        .map(char::from)
        .collect()
}
```

ERROR.RS

```
use std::{fmt, io};

#[derive(Clone, Debug)]
pub struct Error {
    pub kind: ErrorKind,
    pub msg: String,
}
```

```
#[derive(Clone, Debug)]
pub enum ErrorKind {
    ValveNoResponse,
    ValveMismatchedState,
    ValveUnknownStatus,
    MeterNoResponse,
    MeterNoValidData,
    MotorBlocked,
    MotorNotWorking,
    MotorOverCurrent,
    PickError,
    PickFeedBack,
    PickNoConnection,
    PickReadError,
    PickWriteError,
    NoRecovery,
    OutOfRange,
    IoError
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "An Error occured: {:#?} with msg: {}", self.kind, self.msg)
    }
}
impl From<io::Error> for Error {
    fn from(error: io::Error) -> Self {
        Error {
            kind: ErrorKind::IoError,
            msg:error.to_string(),
        }
    }
}
```

THREADS.RS

```
#[derive(Debug, Clone)]
pub enum ThreadCmd {
    FetchData,
    SendData
}
```

PROSPECTS

This work was meant as continued effort for a greater project around IoT-based solutions to sensor-networks and payment solution provision. We have gathered valuable information and experience, extended knowledge and awareness into fields of economy, legal requirements, and technology.

With this knowledge, we feel better prepared for the goal of a serial production of crypto-based water meters. We will continue to make effort to develop and improve the physical prototype, maintain and expand business contacts, and networking in order to bring this project to a fruitful conclusion.
To emphasize on this, fellow colleague Mr. C. Butze is currently working on his bachelor thesis with a more economic focus of the realisation of a SaaS-model, and we are applying for a EXIST founder scholarship.

As far as scientific prospects, we express strong hopes that the european and especially german legal requirements for smart, or IoT-, devices will be overlooked in the near future. We think that a digitization of large scale service providers, like garbage, power, water, internet, and public transport can bring tremendous increase of living quality, while freeing important workforce and providing necessary reliability and security.

The projects findings may also help raise knowledge in the field of sensor networks based on cryptographical protocols, especially the IOTA network. We encourage to apply knowledge gained in projects to help explore the use-case oriented, practical applications of this technology.

SOURCES & CITATIONS

[1] Internship Report & Kolloqium on “Implementation of blockchain technology on sensor-based networks”, F. Rosenkranz, 2022

[2] Bitcoin: A Peer-to-Peer Electronic Cash System
Satoshi Nakamoto, www.bitcoin.org
2008

[3] IOTA media and asset library
IOTA Foundation, assets.iota.org
as of 01.01.2023

[4] What is Chrysalis?
IOTA Foundation
as of 01.01.2023

[5] Protocol Improvements
IOTA Foundation
as of 01.01.2023

[6] Path to Chrysalis
IOTA Foundation
as of 01.01.2023

[7] v2.iota.org

[8] FPC-BI: Fast Probabilistic Consensus within Byzantine Infrastructures, Rev. 3
S. Popov, W. J. Buchanan
Journal of Parallel and Distributed Computing, Volume 147, pages 77-86
January 2021

[9] Robustness and efficiency of leaderless probabilistic consensus protocols within Byzantine infrastructures
A. Capossele, S. Mueller, A. Penzkofer
ArXiv
November 2019

[10] Fast Probabilistic Consensus with Weighted Votes
S. Müller, A. Penzkofer, B. Kuśmierz, D. Camargo, W. J. Buchanan
Proceedings of the Future Technologies Conference (FTC) 2020, Volume 2 pp 360–378
November 2020

[11] IOTA 2.0 Research Specifications, Introduction
IOTA Foundation
as of 01.01.2023

[12] IOTA 2.0 Research Specification, Mana
IOTA Foundation
as of 01.01.2023

[13] Einrichtung einer intelligenten Ausleseeinheit für Verbrauchsmeßzähler,
Diplomarbeit of C. Bories
March 1995

[14] About IOTA Streams
Iotaledger/streams/README.md on Github
as of 01.01.2023

[15] US Patent 4689740: Two-Wire Bus-System Comprising A Clock Wire And A Data Wire For Interconnecting A number Of Stations
Google Patents, patents.google.com/patent/US4689740A/en
as of 31.12.2022

[16] UM10204: I^2C-bus specification and user manual
Rev. 7.0 - 1 October 2021
NXP B. V. 2021

[17] A Protocols for Packet Network Intercommunication
V. G. Cerf, R. E. Kahn, IEEE
Vol Com-22, No 5 May 1974

[18] TCP/IP Internet Protocol
The History of the Internet Web Archive
web.archive.org/web/20180101082256/https://www.livinginternet.com/i/ii_tcpip.htm

[19] OSI-Modell, image copyright:

Beckhoff Automation GmbH & Co. KG
https://infosys.beckhoff.com/index.php?content=../content/1031/tf6310_tc3_tcpip/84246923.html&id=
[20] Telematics Chapter 8: Transport Layer
J. H. Schiller, Freie Universität Berlin
as of 31.12.2022
[21] NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition
NIST.gov website
October 2012
[22] Keccak Implementation Overview Rev. 3.2
G. Bertoni, J. Daemen, M. Peeters, Gilles v. Assche, R. v. Keer
May 2012
[23] Federal Information Processing Standards Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions
Information Technology Laboratory, NIPS
August 2015
[24] The Keccak SHA-3 Submission, Rev. 3
G. Bertoni, J. Daemen, M. Peeters, Gilles v. Assche
January 2011
[25] IBM Unveils 400 Qubit-Plus Quantum Processor and Next-Generation IBM Quantum System Two
IBM Newsroom
November 2022
[26] State of Quantum Computing: Building a Quantum Economy
Isight Report, World Economic Forum
September 2022
[27] Announcing Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms
Computer Security Resource Center, NIST
December 2016
[28] Post-Quantum Cryptography: Overview
CSRC, NIST
as of 01.01.2023
[29] The United Nations World Water Development Report 3
The United Nations Educational, Scientific and Cultural Organization (UNESCO)
2009
[30] Studie Wasserverbrauch und Wasserbedarf, Teil 1: Literaturstudie zum Wasserverbrauch - Einflussfaktoren, Entwicklung und Prognosen
DI Dr R. Neunteufel et. al.
Nov. 2010, p. 28, 59f,
[31] The United Nations World Water Development Report 2022: Groundwater: Making the invisible visible.
UNESCO
2022
[32] Water resources across Europe - confronting water scarcity and drought
European Environment Agency (EEA), Kopenhagen
2009
[33] Wohnungen nach Baujahr
Statistisches Bundesamt, 2018
as of 01.01.2023
[34] www.hyperledger.org/about/members
as of 31.12.2022
[35] Data Sheet Bmeters GSD8-RFM Rev. 22.1
Bmeters Srl Italy
as of 02.01.2023
[36] IOTA Wiki
IOTA Foundation
as of 01.01.2023
[37] Raspberry Pi Hardware Documentation: GPIO and the 40-Pin Header, Raspberry Pi Foundation
https://www.raspberrypi.com/documentation/computers/raspberry-pi.html
as of 01.01.2023
[38] The Rust Programming Language Book
The Rust Foundation 2016
as of 27.12.2022
[39] tokio.rs/tokio/tutorial
as of 31.12.2023
[40] docs.rs/tokio/latest/tokio

as of 31.12.2022
[41] https://docs.rs/tokio/1.5.0/tokio/sync/index.html
as of 31.12.2022
[42] RFC 1122: Requirements for Internet Hosts -- Communication Layers
Internet Engineering Task Force (IETF)
October 1989
[43] M-Bus Wired Documentation EN13757
https://m-bus.com/documentation
as of 31.12.2022
[44] Streams Specification
Revision 1.0 A
IOTA Foundation

FIGURES

ABBREVIATIONS

ACK	Acknowledge Bit
API	Application Programmer Interface
ATX	Atomic Transactions
BTC	Bitcoin
CC	Cryptocurrencies
Chrysalis	IOTA 1.5 Update
Coordicide	IOTA 2.0 Update
DAG	Directed Acyclic Graph
DDML	Data Definition and Manipulation Language
DLT	Distributed Ledger
EEA	European Environment Agency
ETH	Ethereum, Cryptocurrency
EXIST	German founder scholarship
FIAT	Currency not backed by physical commodities, usually government-issued
FIPS	Federal Information Processing Standards
FPC	Fast Probabilistic Consensus
GPIO	General Purpose Input/Output
HIGH	High voltage signal
IoT	Internet of Things
IOTA	Cryptocurreny
LOW	Low voltage signal
NACK	Not Acknowledge Bit
NFT	Non-fungible token
NIST	National Institute of Standards and Technology
ONE	Electrical representation of a computational 1
OSI	Open Systems Interconnection
PoS	Proof of Stake
PoW	Proof of Work
PP	physical pin
PRNG	Pseudo-random number generator
RFC	Request for Comments
RP	Raspberry Pi
RTU	Remote Termnial Unit
Saxeed	Saxonian founder network
SCL	Serial Clock
SDA	Serial Data
SHA	Secure Hash Algorithm
Shimmer	Cryptocurrency
SSH	Secure Shell
UNESCO	United Nations Educational, Scientific and Cultural Organization
URTS	Uniform Random Tip Selection
UTXO	Unspent Transaction Output
WEF	World Economic Forum
WHZ	Westsächsische Hochschule Zwickau
WWZ	Wasserwerke Zwickau
ZERO	Electrical representation of a computational 0
ZIM	Zentrales Innovationsprogramm Mittelstand; Scholarship program