RheinMain University of Applied Sciences

Faculty Design Computer Science Media
Master's degree in computer science
(Smart Systems for Man and Technology)

Master's Thesis
for obtaining the degree
Master of Science – M. Sc.

Self-governing systems
A decentralized revolution

Submitted by Harald Heckmann
on 14th December 2020

Referee: Prof. Dr. Marc-Alexander Zschiegner
Co-Referee: Prof. Dr. Steffen Reith

For Katharina Heckmann, Thorsten Knoll and Leonore Knoll.
Notwithstanding the circumstances, those wonderful people trusted and supported me.
I am grateful for the loving space you created, in which I could freely prosper.

Erklärung gem. BBPO 4.1.5.4 (3):

Ich versichere, dass ich die Master-Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum, Unterschrift Studierende/Studierender

Verbreitungsformen:

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Arbeit:

Verbreitungsform	**Ja**	**Nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger	**X**	
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	**X**	
Veröffentlichung des Titels der Arbeit im Internet	**X**	
Veröffentlichung der Arbeit im Internet	**X**	

Ort, Datum, Unterschrift Studierende/Studierender

Abstract

Significant advancements in decentralized storage and computation technologies, decentralized consensus-based distributed ledger technologies and artificial intelligence have created new but largely unexplored possibilities. This work evaluates how those technologies can be combined to create a decentralized platform, that is completely controlled by the community and whose purpose is solely defined by the community. The system permits the bundling of power in form of central authorities and equally spreads the power over the community, the only requirement for participation being the possession of a computer with a webcam. In addition, financial wealth must not lead to an higher impact. When launched, the systems purpose is not defined. The default features of the system evolve during time depending on the ideas of the community and truly democratic processes. The system is able to reward users for participation with an intrinsic cryptographic currency. This should encourage the users to share ideas and concerns and in addition enables the possibility to spawn projects, which are realized and operated by the community, whereupon the participants are rewarded by the system. Whatever is decided by the community and created within the system, is paid by the system and owned by everybody, it becomes a common good. Hence, the community shapes the system and defines its purpose so that it serves the community according to their needs.

Zusammenfassung

Maßgeblich Fortschritte in dezentralisierten Speicher- und Berechnungs-Technologien, dezentralisierten konsensbasierten verteilten Hauptbuch-Technologien und künstlicher Intelligenz haben zu neuen, jedoch überwiegend unerforschen Möglichkeiten geführt. Diese Arbeit evaluiert wie diese Technologien zu einer dezentralisierten Plattform kombiniert werden können, die vollständig von der Gemeinschaft kontrolliert wird und deren Zweck ausschließlich von der Gemeinschaft bestimmt wird. Das System verbietet die bündelung von Macht in Form zentralerer Autoritäten und verteilt die Macht über die Teilnehmer des Systems gleichermaßen, wobei die einzige Vorraussetzung zur Teilnahme das Besitzen eines Computers mit einer Webcam ist. Weiterhin darf finanzieller Wohlstand nicht zu einem höheren Einfluss führen. Der Zweck des System is zum Start zunächst unbekannt. Dieser entwickelt sich über die Zeit in Abhängigkeit von Ideen der Gemeinschaft und echten demokratischen Prozessen. Das System ist in der Lage, Benutzer für die Teilnahme mit einer intrinsischen kryptographischen Währung zu belohnen. Dies soll Benutzer dazu anregen, Ideen sowie Bedenken zu teilen und ermöglicht weiterhin Projekte zu erzeugen, die von der Gemeinschaft durchgeführt und betrieben werden, wobei die Teilnehmer für ihre Leistungen von dem System belohnt werden. Welche Entscheidungen die Gemeinschaft auch treffen mag und was innerhalb des System erzeugt werden mag, wird von dem System bezahlt und gehört anschließend jedem, es wird ein Gut der Gemeinschaft. Folglich formt die Gemeinschaft das System und definiert dessen Zweck, so dass es der Gemeinschaft entsprechend derer Bedürfnisse dient.

Contents

1. Introduction

This thesis was created with the hope that during its creation an approach for a decentralized system is found, that is controlled and formed by a community and that leads to a greater collection of common goods. The system must not allow a greater influence depending on financial wealth. Hopefully this thesis can assist communities, desirably even humanity as a whole, to organize and manage themselves more effectively by including every opinion and excluding egocentric interests as much as possible.

1.1. What is this thesis about?

The major goal of this thesis is to research (the algorithmical and technical requirements) and deliver a design and a "Proof-of-Concept" (PoC) implementation of a software product, which offers the following functionality:

- Decentralized, immutable, permissionless and trustless ledger, with decentralized storage and computation capabilities, which creates and uses a native cryptocurrency.
- Decentralized creation and management of flexible "Decentralized Autonomous Organizations" (DAO) by the community.
- Offers an intrinsic cryptographic currency that is ultimately used for rewarding the community for participating in the system.
- Offers an identity and onboarding logic.
- Offers any member of the community to participate in projects funded by the intrinsic cryptographic currency.

In summary a software product is researched and implemented as a PoC implementation, which offers the decentralized creation and management of organizations, which are ultimately controlled and operated by the community itself. Ideally this leads to disbandment of structures of power and might, due to the fact that the power over the creation of DAOs, projects and funding of work is shifted from a small group of users

to a whole decentralized community. Since a single group of people will not be able to have an unevenly high impact and profit for themselves, the possibility opens that humanity begins to focus on decisions that profit the whole organization, society or even life as a whole. To illustrate where humanity is currently and what humanity can potentially become by using this system, the term of a superorganism is defined and its properties and the corresponding properties humans already do have and don't have are presented now.

Definition 1.1.1: Properties of a superorganism

1. Cooperative coalescence of many organisms.
2. The performance of the superorganism transcends the performance of each individual by far.
3. It is interconnected, autonomous and self-organizing.
4. The highest goal of a superorganism is to preserve itself, while living in harmony with any other organism and its environment. Life is flourishing in the presence of a superorganism, because its action is always conducive in regards to life.

Humanity does not yet form a superorganism, because not all of the prerequisites mentioned in definition 1.1.1 are fulfilled. Some organizations formed by humans do fulfill all requirements though and may therefore be called a superorganism, based on this definition.
Condition 1 and 2 are fulfilled. We have many cooperative organizations of many organisms of our species. The performance of those organizations does outreach the performance of each individual by far. For example, an organization which produces "Central Processing Units (CPU)" is, by cooperation of each individual within this organization, able to produce a product that no member of the organization is able to grasp in its entirety. However, humanity as a whole can't be regarded as a superorganism, because it is neither autonomous nor self-organizing. Most of the time humanity delegates all of the power to a minority of people. The worldwide behavior of human beings does not mirror the properties mentioned in number 4 of definition 1.1.1. Looking at the world at the time of the publication of this work, humanity is working hard to preserve itself, but not while living in harmony with any other organism and its environment. Life is not flourishing around humanity currently, life is dying and its colorful variety is being

diminished in the presence of humanity.
This work and the PoC software product that is created should help humanity to obtain property 3 of definition 1.1.1. The decentralized interconnected nature of the software and the tools it offers to autonomously self-organize should solve the remaining issues. It can't be guaranteed that property 4 can be obtained automatically by using this software, it does ease the evolution towards obtaining this property though. This is due to the fact, that structures of power are inherently impossible due to the nature of the system. Additionally, and only in combination with the impossibility to bundle power, the collective management of community funds might lead to another way of thinking about our projects and therefore how we spend our time and ultimately our vital force. Instead of profiting for ourselves, we might start to think about projects and in unison choose projects, which improve the quality of life for the whole community.
There are superorganisms formed naturally outside the domain of humanity. Bees are a great example, they reflect all four properties of a superorganism in their way of living.

1.2. Which critical problems are examined in this work?

One major problem this thesis does examine is the task to self-govern. How can a system allow anyone to participate and to integrate themselves in the evolution of this system without having a central authority? Not much time has passed since decentralized consesus protocols have been technically realized, but their potential as a fundamental technology for self-governing systems has yet to be explored. This thesis does answer the question about when a system can use a global (in the context of the system) consensus mechanism and when a fallback solution to some kind of hierarchical structure is required.
When does the system issue new coins of the intrinsic cryptographic currency and how much does it issue? This is also a central question of this thesis, because it helps a community to become independent from investors and therefore choose the next projects without an influence, that aims to profit. The power money has and therefore the influence of someone who owns a lot of it, will potentially be nullified.
If a system allows itself to be governed by a decentralized community, then how does it impede fraud? It is an important goal to obviate the necessity of powerful delegates whenever its possible, therefore the system does often make use of decentralized vote and evaluation mechanisms. How can a system counteract against fraud votes, for example from bots? If parts of a community decide, how to make sure that the members involved do not act in their own interest? How to encourage and reward the detection of fraud?
Another problem to solve is, how to bootstrap the system. How are the first blocks created? How to verify identities in the system, if no verified identities are available to carry out this task? How to finance the development of the platform?

1.3. What are the novelties?

An organically growing digital decentralized organization, that is controlled and operated by the community, where everybody is free to enter and participate in the roles they desire, does not yet exist in that form. Decentralized consensus mechanisms in combination with thoroughly selected, but still mutable consensus rules, allow the system to offer this functionality. The fact that the community controls and builds the system leaves no room for intermediaries and wealthy, to influence and profit. As child organizations and projects are worked out by the community, the requirement for people that can fulfill the derived tasks is inevitable. The system encourages its members to apply for those tasks and does pay them fairly, based on decisions by the community and previous data. Another feature is the decentralized operation of the products/services which were created by the community. Projects also allocate funds, that exclusively serve for paying community members who support the system with storage capacity and computation time. Since the system also pays the operation of projects, all products and services can freely be used by the members of the system. The products and services therefore become common goods. The exceptional task of dreaming visions of how the community of tomorrow should look like and what it is missing to get there, is the only task that is paid percentaged, if the community decides that it wants to evolve in this direction (and consequently votes for realization of this idea). This should encourage everybody to think about what the community really needs. Any other work or form of participation is paid using a fixed value.

1.4. Why now and not earlier?

Much progress in technology during the past years lead to interesting results, some of which can be combined to result in a project like this. One major technological achievement is the development of decentralized, permissionless, trustless and immutable systems. Those serve as the fundamental building block of a decentralized self-governing system. If the system can gain or lose members at any time and does not rely on central authorities to make and implement decisions, it must offer some decentralized and trustless consensus mechanism. With the publication of the bitcoin paper, this problem was solved[Nak09]. In addition, bitcoin introduced a new data structure, the blockchain, which in combination with the consensus mechanism enables immutable storage. The blockchain contains every state the system was in, and since bitcoin is a financial system, it also includes every transaction of bitcoin, which at some point in time were included during the execution of the consensus mechanism. This technological achievements are fairly new and are indispensable to the system that is researched in this work. A decentralized community cannot manage itself without a consensus

mechanism. To understand the evolution of the system and detect undesirable behavior, a public and immutable history is required. Since this system should allow anybody to participate, it must be permissionless. Since anybody can participate, it must be trustless, such that every state changing action is verifiable. All those properties came with bitcoin and are used in the system this work examines.

Another important technological possibility are self-sovereign identities. People can present themselves as anything they like in different scenarios. They can present themselves as a lawyer in one moment and as a supporter of a specific movement at another moment. Rarely will a lawyer present his personal believes during the time he or she plays the lawyer. This is a simple example of how a person carries multiple identities, which are used in different scenarios. However, an identity card is certified by the state and therefore the person will have a hard time during trying to fake the name for example, if somebody demands for the identity card. The same mechanism was digitally implemented. Users can create as many "Decentralized IDentities" (DID) as they like. They can collect signed information from other identities, for example that they master a specific competence, which are bound to one of their identities. This information was issued in form of a signed certificate, which is stored in a data container named "Verifiable Credential" (VC). This way they can use different identities for different occasions. An permissionless system must offer some kind of digital identity management system, otherwise there is currently no way to ensure that people don't vote multiple times for example or to verify that specific properties which are presented by the identity really exist.

Two other important technological components are decentralized file storage and decentralized computing software products. They allow every community member, regardless of how much storage capacity or computation time they have available, to participate and earn a reward. Decentralized storage tools, like torrent, exists for quite some time. Newer protocols like "InterPlanetary File System" (IPFS) are an improvement in regards of speed, security and functionality. Furthermore, cloud computing is being offered for non-expert users by amazon "Elastic Compute Cloud". Although the service is provided by a central authority, the way the system allocates and deallocates CPUs from spatially disconnected computing centers can be applied to this work. By using this technologies, the community can participate in offering services, earn a reward by doing so and at last stop industry giants to create monopolies.

2. Fundamentals

2.1. Blockchain framework: Substrate

This section offers an introduction into blockchain technologies with focus on the Substrate blockchain framework.

2.1.1. Summary: Blockchain technologies

The term blockchain is often used for a complete decentralized consensus-based system. De facto the blockchain is a linked list, where each element within the linked list contains the hash value of its ancestor. Therefore the removal of one element invalidates all its children elements, because they (directly or indirectly) contain its hash value. The term blockchain originated in the first decentralized financial transaction system Bitcoin[Nak09], where the blockchain is used to record all financial transactions. Bitcoin is a decentralized, (probabilistic) immutable, permissionless and trustless system. It is decentralized, because no central authority stores data or dictates the state changes. It is (probabilistic) immutable, because once transactions are recorded in the blockchain, they cannot be changed due to the fact that every element references the hash value of its ancestor. In fact it is only probabilistic immutable, because in practice the probability to replace a block within the blockchain only is lowered drastically with each block, reaching almost 0% after six blocks (see section 11, "Calculations" in the original bitcoin paper[Nak09]). It is permissionless, because anybody can participate in the system and freely choose the roles in the system. It is trustless, because the source-code is open source and every state change in the system is verifiable by anyone. A blockchain-based system like that will be called a "dipt-blockchain" (decentralized, immutable, permissionless and trustless blockchain) in this work.
One of the greatest novelties that Bitcoin introduced is, that it achieved byzantine fault tolerance. A byzantine fault tolerant system is tolerant against failures of participants in a decentralized network. The term byzantine fault tolerance is derived from the byzantine generals problem[LSP82]. Bitcoin achieved this by introducing a novel consensus-mechanism, called Proof-of-Work (PoW). PoW is fault tolerant until 51% of the systems participants fail. In Proof-of-Work, every participant competes in

solving a computationally difficult problem. The system adjusts the problem difficulty depending on the total computation power of the participants, such that the result is approximately found by one participant within 10 minutes. The winner of the competition is selected to author the next block, id est determine the next transactions that are executed and therefore determine the next state of the system. The syntax and semantics of the block must obey consensus rules, which everybody knows and uses to verify each block. If the block does not follow those rules, it is omitted. Solving the problem requires a lot of computation power, which in turn requires electricity. To motivate members to honestly participate, a reward in form of a digital currency (also called cryptographic currency) is credited to the block author. The currency is bound to an address, which is derived from a public key, and only a valid transaction that was signed with the corresponding private key does move the currency to the same or another public key, once it is included in a block.
After Bitcoin operated successfully for four years, a new proposal arrived, which described a generalized dipt-blockchain called Ethereum[But20][Woo20]. Ethereum includes a virtual machine, which is (quasi) turing complete. It is only "quasi" turing complete, because executing an opcode consumes some of the intrinsic cryptographic currency that was supplied to the call, and once the supplied funds are consumed, the execution is discontinued. Ethereum introduces explicit accounts, that also can contain code that is executable by either a local call to the ethereum virtual machine (read only) or a call instructed through a transaction (read-write). This feature turns Ethereum into a decentralized computer.

2.1.2. Design

The Substrate blockchain development framework was created as a side project of a greater vision: The Polkadot network, a vision for a heterogenous multi-chain framework[Woo17]. Polkadot is essentially a dipt-blockchain, that orchestrates a mutable network of dipt-blockchains, which contribute their unique features to the ecosystem in a trust-free manner. During the development of Polkadot, the development team recognized that much functionality used in such a system can be generalized. This lead to Substrate, a blockchain development framework that offers the developer to build upon a basis of generalized functionality. This way the developers can focus on implementing the actual business logic, that is part of the "State Transition Function" (STF) of the system, without having to worry about core functionality like networking, database management, client functionality, etc. Figure 2.1 shows the separation of core logic and business logic into a node and a runtime project respectively. A module that extends the STF is called pallet.

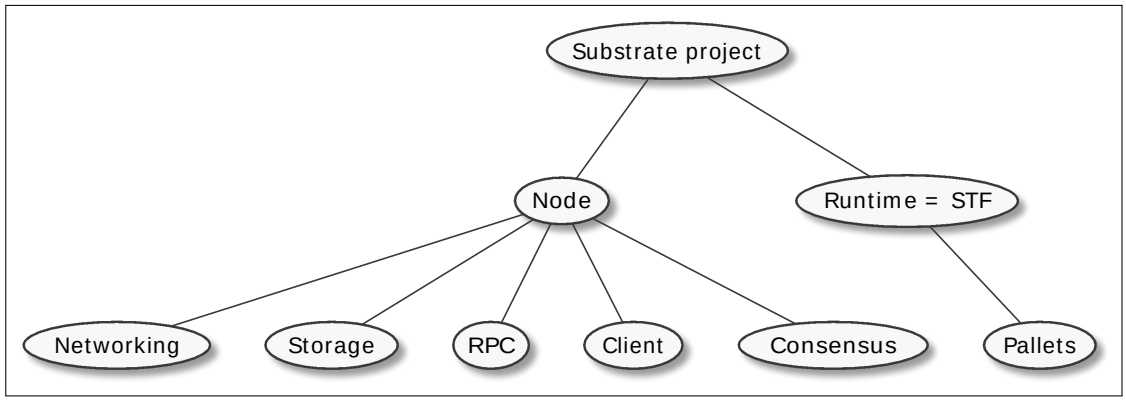

Figure 2.1.: Separation of core and business logic in Substrate

2.1.3. FRAME: Modular extension of the state transition function

"Framework for Runtime Aggregation of Modularized Entities" (FRAME) consists of runtime modules (pallets) and support libraries that ease runtime development. Pallets are modules within FRAME, that describe a part of the STFs logic. A runtime developer can use the support libraries to access core types (e.g. account id or hash type) and system-critical storage items (e.g. block hash or events). Support libraries also offer the developer functionality to generate errors, events, the module logic for dispatachable calls (for example calls that are instructed within a transaction) and to generate a storage layout with functions to modify it. In addition, macros help integrating the pallet into the runtime. Within a pallet, other pallets public function can be called. For example, a pallet that manages account balances of the intrinsic cryptographic currency could be invoked within another pallet to generate, destroy or move a balance.

2.1.4. Cryptography

The two major cryptographic primitives used in most dipt-blockchains and Substrate, are cryptographic hash functions and digital signatures based on public key cryptography. In Substrate, hashing is used to determine the block hash, which represents an unique id based on the contents and which is referenced by a child block if it exists. It is also used in storage to create hashmap, although the hash function used can also be non-cryptographic, if desired. Besides those applications, hash functions are arbitrarily usable within pallets. The default cryptographic hash function is blake2b[SA15]. As a non-cryptographic hash function, xxHash is used, which offers a major speed advantage in comparison to blake2b[1]. Generally, Substrate allows to used any hash function, that implements the "Hasher" trait (interface) provided by the framework.
Digital signatures are primarily used by any user to sign transactions and by block

[1]Hashing speed comparison at `https://cyan4973.github.io/xxHash`

authors and validators to sign messages shared between them. By default it uses the ECDSA[JMV01] signature scheme based on the secp256k1 curve, which is also used in Bitcoin and Ethereum. In addition, it provides implementations for Ed25518 EdDSA[JL17] signature scheme on curve Curve25519, which is optimized to increase speed without compromising security. Last, it also provides a Schnorr signature scheme[Sch89], which also uses the curve Curve25519. Schnorr signatures are generally more resistant to misuse, offer the powerful feature to aggregate signatures and generally are better fit for hierarchical deterministic key-derivation. This come at the cost though, that the signature does not provide information to reconstruct the public key associated to the private key that was used to create the signature. Since the ecosystem binds accounts to addresses which are ultimately derived from a public key, a transaction that was signed with the Schnorr signature scheme must include the public key that signed the transaction in addition to the signature. So for Schnorr signatures, in addition to 64 bytes of signature data, 33 bytes of compressed public key data must be included as well, which compared to ECDSA demands for 50% more space for signature data. Substrate also allows to use any public key cryptography based digital signature algorithm, the only requirement being that it implements the "Pair" trait.

2.1.5. Storage

Substrate uses a modified merkle partricia trie to store data, which is a key-value storage data structure that operates on Base 16 keys. It is a combination of two trees: A merkle tree[Mer88] and a patricia trie[Mor68]. The patricia trie is a key-value based search trie and is used to navigate through the trie to find, store, update and delete data. The merkle tree is used to efficiently verify the integrity of data. Two merkle tries that represent different datasets will always have different root hashes, therefore two equal root hash values represent the same dataset. This mechanism is used in Substrate for simple payment verification and for verification of the internal storage. The Substrate state is equal to the internal storage, therefore matching two root hashes successfully on equality proves, that the states of the systems the two root hashes originate from are equal. Since every storage root hash is stored in the block header of every block, every full node can easily verify at any point in time, if the state of the node matches the canonical state. Simple payment verification is the process of verifying the inclusion of a transaction into a block, without fetching the transactions included in the block. This is achieved by a merkle proof over the extrinsics, whose root hash is stored in the block header and should match the root hash of the merkle proof.
The default patricia trie is a radix trie with radix 2, but Substrate uses a patricia trie with radix 16 in their modified merkle patricia trie implementation, such that each node can branch up to 16 times. It has three different types of nodes to navigate through

the tree: A branch node, an extension node and a leaf node. The type of node used depends on the the data that is stored in the trie and the position of the hexadecimal value in the storage key. When multiple data entries exist, whose keys start to differ at position p, a branch node is inserted that branches the trie to represent a path for each of those different key variants from position p onwards. For example, k1 is "1C" and k2 is "EB" and k1 and k2 are the only keys. In this case, a branch node would be inserted, that branches into "1" and "E". If some keys continue identically before branching, the extension node is used to skip through a number of common hexadecimal values. For example, k1 is "59EA1C" and k2 is "59EAEB", and k1 and k2 are the only keys. In this case, an extension node would be inserted, that indicates to skip "59EA", because a comparison of every single hexadecimal value up to that point is not necessary. At last, a leaf node is inserted that contains the remaining hexadecimal values of a key and the associated data. Every node contains the hash value of its children nodes, which turns the modified patricia trie into a modified merkle patricia tree. Figure 2.2 depicts the complete modified merkle patricia tree that results for the two keys k1 = 59EA1C and k2 = 59EAEB, with the associated data "Hello" and "World" respectively.

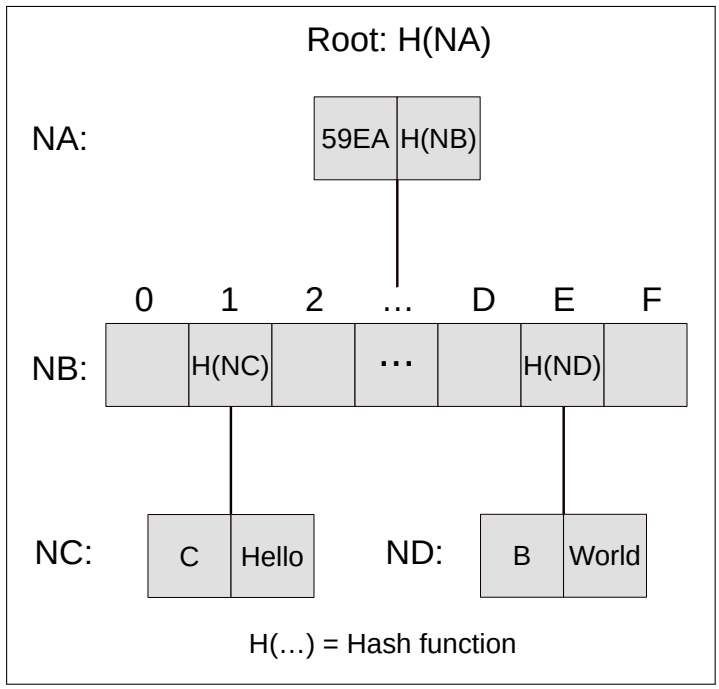

Figure 2.2.: Simplified view on the merkle patricia storage trie

2.1.6. Extrinsics

An extrinsic is data that is included from outside the chain into the chain. Signed transactions, unsigned transactions and inherents are default extrinsics used in Substrate. A

signed transaction is the classical and only extrinsic used in most other dipt-blockchains. The nodes can verify its validity by checking the signature and verifying that the associated public key is authorized to instruct the execution of the transaction, for example by checking if the public key owns enough funds to pay the transaction fees.
An unsigned transaction is more complicated to handle. It is missing a signature and an associated public key, which normally would be used to validate the transaction. Since the nodes are not able to check the transaction fees, a mechanism to prevent spam is missing. In addition, no nonce is supplied, which implies a vulnerability against replay attacks. Therefore unsigned transactions must always include another mechanism to determine the transactions validity and obviate the possibility for spam or replay attacks.
Inherents are unsigned data that the block author includes into the block. They are not gossiped through the network like transactions. Their validity is solely determined by the fact, that a sufficient amount of validators have checked them and came to the conclusion, that they are reasonable. For example, a block author can include a timestamp into the block. Every validator does compare the timestamp to the local clock and if the timestamp would surpass the margin of error, the block would be rejected. A validator is a user that participates during the block authoring process and helps to determine valid blocks.
Extrinsics offer the developer great flexibility to define the structure of the data and a mechanism to determine their validity. While being a powerful feature, it is also a feature that deserves much consideration, because the flexibility brings an high risk of creating new vulnerabilities.

2.1.7. Weights

Weights are a concept used to measure the execution time of a function. The longer the execution time, the heavier the function is. One unit of weight is defined by one picosecond of execution time on the following reference system:

- CPU: Intel Core i7-7700K
- RAM: 64 GB (type and clock speed not specified)
- Hard drive: NVMe SSD (read/write speeds not specified)

The weight is used to calculate transaction fees and to calculate the execution time required to process the function invocations within a transaction, which is also used as a decisive factor during the inclusion of the transaction into a block. This is due to the objective to keep a constant block production time. The developer of a dispatchable function is responsible for supplying a reasonably precise maximum weight consumption.

2.1.8. Forkless runtime upgrades

In a classical dipt-blockchain a client update is a difficult problem. When the update includes changes to the consensus rules or STF that lead to a different outcome regarding block verification, a (hard) fork occurs. An hard fork is the situation, during which some participants accept a block and others do not accept it. In this case, one group includes the block in their canonical blockchain and the other does not. From that block onwards, including the block itself, the chain differs for both groups. This is called a hard fork. Backwards-compatible client updates lead to a soft fork, the software differs, but both clients decide for the inclusion or omittance of a block in unison. Consequently, the chains are identical in both clients.
Substrate offers a new solution to update the clients STF without inducing a hard fork in any case, even if both versions, the updated and the old versions, come to different conclusions regarding the verification of blocks. The Substrate runtime is compiled in two output formats. One output format is called native and results in binaries that are directly executable on the architecture they were compiled for. The other format is called Webassembly (Wasm). Wasm is a portable binary-code format that includes a load-time-efficient stack machine. It describes a (probably) memory-safe, sandboxed execution environment. It is designed to maintain the versionless and backwards-compatible nature of the web. Substrate uses those properties to introduce a novel feature, forkless runtime upgrades.
Whenever the runtime is updated through the `set_code` call of the runtimes system library, a Wasm runtime is stored in the storage. Since the storage root is equivalent to the systems state and included in the block header, the upgraded runtime is also part of the state. Whenever a native Substrate (non-light) client starts or fetches a block that contains a call to `set_code`, it compares the version of the Wasm runtime stored in the local storage to its native runtime version. In case the Wasm runtime is more recent, the client uses it instead of the native runtime. This way different local client versions will never come to a different conclusion during block verification, because all clients use the same STF, either by using the most recent runtime as a native binary file or by fetching the most recent runtime from the canonical block chain. Updating the native runtime is recommended, because it is more efficient compared to a Wasm runtime.

2.2. Decentralized storage

InterPlanetary File System (IPFS) is a decentralized file sharing platform. One goal of IPFS is to provide an efficient system to realize a distributed file system[Ben14]. In this section, some of the unique features IPFS offers are described.

2.2.1. Content addressing

Usually local and remote data is addressed by supplying a path, for example a web address or a path on the local hard drive. In IPFS, data is addressed based on its content. By using a cryptographic hash function, every distinct file practically is represented by a different hash value. It has only practically a different hash value than another file, because the input size is arbitrarily large and the digest size (hash size) is finite. Therefore a collision is inevitable according to the pigeonhole principle, given that atleast `digestsize` $+ 1$ different input values are hashed. Since minor changes in the input value result in a completely different hash value, the number of distinct input values that must be hashed to find a collision is so vast, that in practice a collision is not found. According to the birthday attack[Gup15], which is derived from the birthday problem[BH07], the number of hashing attempts to find a collision for two arbitrary preimages with a probability p for a digest size of H is:

$$n(p; H) \approx \sqrt{2 \cdot H \cdot ln\frac{1}{1-p}} \tag{2.1}$$

To find a collision for a cryptographic hash function with a digest size of 256 bit with 0.1% probability, the number of distinct data samples that must be hashed totals to:

$$n(0.001; 2^{256}) \approx \sqrt{2 \cdot 2^{256} \cdot ln\frac{1}{0.999}} \approx 1.522 \cdot 10^{37}. \tag{2.2}$$

For a 50% chance to find a collision, the number of distinct data samples that must be hashed totals to approximately $4 \cdot 10^{38}$. So practically, no collision should be expected and every piece of data has its own unique hash value. Therefore addressing data by its hash value is a practicable concept.
IPFS encodes the hash value into a base-58 encoded value with additional data. The result is called “Content Identifier” (CID). The CID is used in IPFS to fetch specific data. Section 2.2.2 shows, how the hash contained within the CID is calculated and how it is embedded into a data structure to gain some essential advantages.

2.2.2. Merkle Directed Acyclic Graph

IPFS uses “Merkle-Directed Acyclic Graphs” (Merkle DAGs), which are unbalanced merkle trees that permit multiple ancestors. IPFS splits files into chunks of a predefined (but configurable) size. Every chunk is hashed and is assigned its own CID based on the hash. The chunks together with their corresponding CIDs are vertices in the DAG that do not have a single child vertex. All the CIDs of the chunks of a file are hashed together, which result in the CID of the file. In case the file size is smaller than the chunk size, both are equal and represent one vertex in the DAG. How would a folder look

like in this representation? The CIDs of the files contained within a folder are hashed together and result in the CID of the folder. Once a file was added, deleted or updated, the folder CID changes. In fact, not the Merkle DAG changes, but a completely new Merkle DAG is created. Every vertex in a Merkle DAG is the root vertex of another Merkle DAG. For example, a folder represents the root of a Merkle DAG. Within that Merkle DAG, each file and each chunk of those files are Merkle DAGs on their own.
Using the Merkle DAG to store chunked files and build a file system out of them has two major advantages:

1. The chunks can be distributed over a decentralized network. Upon a request, the chunks can be provided by different participants and can be easily verified. The system can use this feature for example to balance the load and achieve a much higher throughput.

2. The vertices are shared amongst different Merkle DAGs. This eliminates the occurrence of duplicate data in an IPFS node. For example, two different folders that contain the same file are both the ancestor of that file in their corresponding Merkle DAG representation.

2.2.3. Immutability and persistence

Every IPFS node that downloads data from its peers automatically includes the data temporarily in its own Merkle DAG. This way, an higher demand is supported by an higher number of temporarily available providers. The data persists as long as the data is demanded. The provider of an IPFS node can freely decide which data is permanently kept (pinned) in their Merkle DAG.
Files within an IPFS node cannot be changed, they are immutable. Updating a file means adding the changed file to the IPFS node again. Both files have different CIDs, because their hash values changed due to the different contents. From the perspective of the IPFS node, those are different files and both result in different Merkle DAGs. If the files are very similar, there is a chance that some of the chunks are identical and therefore have the same CID. In that case IPFS uses the corresponding vertices of the identical chunks in both files Merkle DAG representation, effectively eliminating duplicates in the node.

3. Design

This chapter describes a design that can be used as a blueprint for the vision delivered in chapter “introduction” (1). It advocates the usage of tools and algorithms described in chapter “fundamentals” (2) and reasons why they are necessary.
In the section “Concept” (3.1) the selection of the main components is presented and justified. Furthermore, some key functionalities that serve as a foundation for more complex architectural additions are described. At last, an overview of the complete system is shown and described.
The next section “Overview of blockchain modules (FRAME-pallets)” (3.2.1) shows a figure that contains the architectural additions in form of single modules to the Blockchain, which ultimately add the functionally that enables the system to be self-governing by way of the community.
After those sections the custom pallets created during this work are described.

3.1. Concept

In this section an high-level view of the design this chapter covers is presented. The major goal is the development of a decentralized platform, in which owners of a computer and a webcam can join the platform, let their physical identity be verified by other verified users and finally propose and vote for future projects as well as participate in their development and operation. Any action is rewarded with an intrinsic cryptographic currency, that the blockchain issues whenever it is necessary.

3.1.1. Overview

The amount of required technology for a system matching the description from section 3.1 is very limited at last. Such a system must provide an upgradable “dipt” (decentralized, immutable, permissionless and trustless) blockchain for reaching consensus about the state of the whole system. It must also provide a decentralized storage where data is redundantly stored, but not as often (in terms of redundancy) as on the blockchain (data is replicated to every full node on the blockchain). Although it is not a technology, it is worth to mention that another key component of the system are its users who

are willing to participate, because they ultimately form and direct the evolution of the system.
In terms of algorithms, the major key algorithms are of cryptographic nature: public-key cryptography or any post-quantum secure analogue that allows authentication, the creation and verification of signatures and for some special scenarios encryption. Authentication and signatures play the main role, they allow members of the system to associate a partial state of the whole system with their accounts and modify this state exclusively on their behalf. This state includes data of any kind, but a classical application is the management of value in form of a cryptographic currency. Encryption is used for applications which run off-chain (that are not evaluated by blockchain nodes) but are coupled with the state of the blockchain in some way, for example for rewards for those who offer storage and computation time for that application. Since encrypted data can only be decrypted by a limited amount of persons, usually one, the storage of such data on the blockchain (which gets replicated on every node which ultimately cannot interpret this data), is unreasonable.
Hashing algorithms are also a key component, since they enable the creation of fingerprints that serve as a reference. Through this mechanism, the blockchain and the decentralized storage are merged to one component. The fingerprints can be seen as pointers (like in the programming language C), they use only 32 bytes of space most of the time, but refer to a specific location in the memory that can contain an arbitrary count of bytes. It should be mentioned that improvident usage of that concept is discouraged. For example, a blockchain can storage these "memory references" (hashes), but in the current state of decentralized storage technologies, it should not query the storage during the execution of any extrinsic. This is due to the circumstance that anything that is executed within a block of the blockchain must be deterministic in terms of (space and) time consumption, otherwise a constant block time cannot be ensured. Current decentralized storage technologies do not carry this property. Another frequent usage of hashes happens during the construction, querying and modification of the blockchain storage, the underlying data structure (a modified merkle patricia trie) of the blockchain storage database uses hashes to navigate efficiently through the nodes of the trie. The typical usage to verify the integrity of data by using a hash is also important. Therefore the hash is not only a pointer to data, it is also a value that can be used to verify that the correct data was received.
Another fundamental algorithm that is required by this system (and still demands for a lot of research) is one that offers extraction of unique physical features of a person that can be used to distinctively map this data to exactly one person. In addition to this, a method is required that allows identified members of the system to reach consensus about the physical identity another person who wants to be mapped to a unique digital identity. This chapters describes the usage and configuration of every compo-

nent mentioned in this section. It also describes a possible solution to the distinctive identity problem, which in the end serves as the foundation for a fair, equal and maybe, sometime in the future, completely free interaction with this system.

Figure 3.1 does depict a very high-level overview of the system. In this figure three components and one participant are depicted. One component is an upgradable "dipt" Blockchain, the other two components are a decentralized storage and a decentralized computation system. The participant is representative for all the different users in the system. The figure shows the relationship between those three components. The blockchain is at its most fundamental level a huge interactive state transition function, which receives its inputs from users, evaluates them, modifies the state accordingly and stores them in an immutable block. The user can interact with it in many ways. Traditionally, the inputs from users were transactions which modified the users balance. In this work, a user can submit any extrinsic to the blockchain. It can be a proposal, a concern, a vote, the result of work, a proof or anything to properly interact with a part of the modules within the blockchain. In exchange, the user can view the complete current and past states of the blockchain and gain rewards for contribution (in form of cryptocurrency). On the other side of the figure it is depicted that a user can exchange data with the decentralized storage. Users can register themselves as storage providers and pin (permanently keep) data. Depending on the storage capacity and bandwidth, an adequate reward is paid by the blockchain. Additionally, users can register themselves to solve specific computational tasks in exchange for a reward.

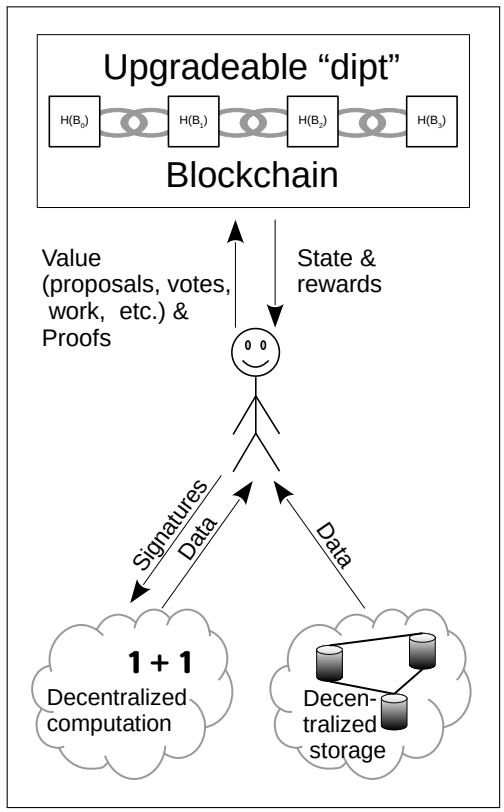

Figure 3.1.: Overview

3.1.2. Blockchain

In the first cryptocurrency, Bitcoin[Nak09], the blockchain was originally designed to lock value and transfer value by the owner. Interestingly there was no state that explicitly mapped an account to a balance, all this was achieved using intrinsically by locking

the value using cryptographic puzzles, “Unspent Transaction Outputs” (UTXOs) , that could only be solved and therefore unlocked and transferred by the true owner (or a phisher or hacker of course).
Ethereum[But20], the next evolutionary step of distributed ledger technologies (DLT), used an universally modifiable state and offered to programmatically extend the state transition function by implementing an invocable smart contract[Woo20]. The current stage of development is reached by Substrate[Fra18] and the Polkadot network[Woo17]. Substrate does not only allow to extend the state transition function by smart contracts, it allows to add modules to the blockchain core itself which modify the universal state transition function. This brings a vast space of new possibilities with it, for example to add logic that is executed before or after the evaluation of a block, apply modifications to the consensus rules, determine the fees or even replace them by another concept, or to issue funds. Since this state transition function is a fundamental part of the consensus rules, normally this would lead to forks, to be precise to multiple chains from that block where the blockchain had multiple rules to interpret a block. By including the new logic into the block that introduced the changes in form of platform-independent WASM (Web Assembly) code, any blockchain node that is operating on old code can fetch and execute the updated code from that block, which at last diminishes the appearance of forks and makes local client upgrades only a matter of performance. The ability to fundamentally change the state transition function and thereby the consensus rules without provoking a fork, is a key feature that enabled the technical description and PoC (Proof-of-Concept) implementation of the system described in this work. Ultimately, this features allow blockchain technology to become morphogenetic (evolve its default features over time). If the features of the blockchain develop over time and are completely in control of the sum of every single organism that is part of that blockchain, the blockchain itself becomes in some way an evolving organism, in which every single user is an integral part of the whole, like a cell in a biological organism.
The Polkadot network is a network of interconnected blockchains, which can communicate with each other and redirect tasks. For example, the KILT[Gmb20] blockchain is part of the Polkadot network and offers identity services. Any blockchain in the network can request the usage of those services. Polkadot is not further examined in this work, nevertheless interesting questions for future work arise. For example, what would a network of morphogenetic blockchains look like? If every member of this network is in some way an organism, can the network be regarded as a society, where every member does overtime find their role and adjust to changes during time?
It is important that the blockchain does carry the “dipt” properties. It must be decentralized, otherwise it can be censored or shut off. If the blockchain is decentralized, its end of life is reached when every member within stops providing its core functions. It must be immutable, otherwise nobody can rely on past decisions. If it would be

mutable, everybody is uncertain at any time about any state change in the past. This includes votes, work contracts and transfer of value for example. The blockchain must be permissionless, everybody must be able to enter and leave the system at any point of time, a selective process of its members mustn't occur at any point in time. This is a mechanism to avoid bias by grouping certain groups of people. Furthermore, it promotes diversity by not only allowing, but welcoming everyone to participate in determination of the direction to evolve. The blockchain must be trustless, any statement that leads to the change of the state must be verifiable. If this property would not be rooted in the system, the whole system collapses. A transfer of value is not verifiable in that case, was it legitimate? Votes would not carry any weight, because it is not transparent if votes were really cast by a single identity. In general, any state transition would be questionable and therefore such a system would not have any use. For the system to be trustless, the code of it must be open-source.

3.1.3. Storage

Data cannot be stored exclusively on the blockchain because it is replicated to every full node in the network. Therefore a separate decentralized storage is required, that reduces the redundancy to a reasonable level. With this design decision one major problem arises. How to verify what amount of storage capacity and bandwidth is supplied by a specific node, to eventually reward the owner appropriately for it? Those are two separate systems, that have to be coupled in some way. If a request for data is dispatched using the IPFS (InterPlanetary File System) protocol, the data is delivered in chunks from multiple IPFS nodes who currently possess parts of the requests data. It is possible to use the HTTP protocol to download data directly from an IPFS node, but this leads to a deficiency by dropping some major benefits of the IPFS protocol:

1. If the host is not available, the requesting party has to find another host on their own. IPFS uses DHTs (Distributed Hash Tables) to automatically find nodes that can provide the requested data.

2. Since data is provided in chunks by multiple hosts (if available), the load is balanced over multiple nodes.

The first idea that arose to tackle this problem was to exchange signatures for data, which can be used to exchange them as a proof for supplying data against a reward. Since the data does most likely come from multiple nodes (because other nodes duplicate data from their peers), it is very difficult to exchange signatures for data. This is due to the fact, that a reward should only be issued if a node preserves the complete data, not just chunks of it. This act is called "pinning" in IPFS. By pinning data, a node ensures that it will keep this data in its entirety within its storage and is able to provide

it completely as long as its pinned.
The solution that is suggested in this work is using a feature of Substrate, the offchain-worker, which allows to compute arbitrarily complex tasks and include the results in the blockchain as soon as the computation is finalized. By doing so, the deterministic block production time is not influenced by an indeterminate complex computation. In this specific scenario, this feature is used to test storage suppliers **??**.

3.1.4. Computation

In contrast to the problems a decentralized storage implicates, the decentralized computation as used in this system is fairly simple. This system uses the term decentralized computation as a collection of decentralized computation units, that offer to atomically execute a task, id est a computation unit is selected to solve a problem as a whole. In a future version of this system, an advanced version that is able to split a problem into chunks, if it is possible, and redirect each chunk to one computation unit to balance the load and allow for possibly faster computations, is desirable. In the one problem per computation unit scenario, the requesting entity can exchange a signature for a result, which can be used to retrieve an appropriate reward for it.

3.1.5. Identity as a basis

The definition and management of identities challenge humanity for quite a long time. First it has to be defined what an identity is. In point of fact, an identity is dynamic and ambiguous. To understand that it is dynamic, one must simply ask oneself whether a person does change over time or not. An identity is composed by specific traits and in fact, those vary over time. Therefore what one might declare as the own identity, might look very different after some time passed. For example, at one point everyone is a child. The identity encompasses to be a child. As time passes, every child eventually grows into an adult. The identity does not encompass to be a child anymore, rather one would associate oneself as an adult at that point of time. To illustrate why it is ambiguous, one can image oneself in different scenarios. For example, during work and during free time with close friends. The traits and look of person often change depending on the environment and context and depending on that, are experienced differently by observers. Therefore an identity is ambiguous. This two properties make it impossible to irrevocably define the identity of someone, the identity is a concept that changes every moment and the presented identity changes depending on the environment and context[Hec20].
To further clarify what identity means in the context of this work, the term is extended and class-divided. First, the term identity on its own is the set of every property a being has at an appointed time. Since this set changes every moment and therefore

already changed when a being thought about it and tried to measure it, it is practically unusable. For example, the nails, the hair, every cell of the body have undergone a transformation from one point in time to another. Everyone can verify this by taking pictures at a discretionary period and comparing them.
The next term is “digital identity”. A digital identity does contain a subset of the identity, preferably slowly changing or even static traits (like a birth place and date). The subset of the identity is mapped to digital data (bits)[Cam04]. A digital identity is used on smart cards for example. In a company, a certificate from the state, namely the identity card, is verified against the person who presented it. If the data matches, a smart card is created that serves as a digital key that is loosely coupled to the subset of traits the state has certified. Using this digital key within the smart card, the person can, for example, authenticate digitally and a system is able to control access based on the permission that were assigned to the person indirectly through the smart card. The problem with digital identities in the context of this work is, that the definition is too broad. It allows central authorities to create a digital identity for another person and often in practice involves that the digital identity is verified against certificates that were issued by authorities in the first place. In addition, for every different environment a new digital identity is created. For example one for work, several for accounts on websites (that use verified and non verified traits), one in university, etc.
In the past years, the term “Self-Sovereign Identity” (SSI) appeared and was further specified[Müh+18]. Looking at the number of papers, a clear trend since late 2015 early 2016 shows rising interest and ideas regarding this concept. A SSI is an digital identity that is created by the user (including machines) that will be associated with the user. Usually a private key is mapped to an identifier. This identity does not contain any further information, but it can be added in form of verifiable credentials, that are a collection of signed claims by other identities. For example, an employer could issue a verifiable credentials containing signed claims that name the activity, duration, etc. It is up to the owner of the identity that receives those credentials to decide which credentials and claims within are presented to other identities, which reflects the ambiguous property of an identity. Credentials can also expire, which reflects the dynamic nature of an identity[NJ20][GH19][TV18]. Recently, the “World Wide Web Consortium” (W3C) started working on a standard that defines the structure and usage of a “Decentralized Identity” (DID) and a “Verifiable Credential (VC)”[Ree+20][SLC19]. Although SSIs are very useful, they are not used in the core design of this work. The implementation of SSIs in a stable future version of this work is highly recommended.
The type of identity required by the system proposed in this work is a Decentralized Digital Identity (DDI). It is a digital identity, but instead of a central authority, a group of users that own validated DDIs are randomly selected out of a decentralized pool of willing users, and are invited to validate a fixed set of traits for another user. If

and only if the group reaches consensus about the identity of the user, a digital identity is created and stored in the immutable blockchain. The system proposed in this work relies on the circumstance that one human being has exactly one identity in the system. This allows every identified user in the system to vote, work, apply for governing rules and more without using stakes (value that is frozen and burned for wrong behavior). By eliminating the need of value in any kind, the power over this system is distributed based on the question whether or not the user is an human being, and not on the question how much value the user owns. How this process works and which traits are used is described in section 3.3.

3.1.6. Intrinsic currency and rewards

In a custom blockchain, such as Substrate, the developers have full control over when and how much currency is issued. This is a major difference to smart contracts, which life in a higher level of abstraction and don't have access to such low-level functionalities like the creation of new coins or the block composition in general. A basic obvious function of coins is too transfer them between different owners as a method of transferring value and to pay transaction fees. The latter usage will eventually diminish in later versions of the proposed system. This is due to the vision, that every human being that was identified as a unique human being in the system, must get access to the system without the requirement to possess anything, but the proof that they are a unique human being. This will require a careful design to prevent "Distributed Denial of Service" (DDoS) attacks. Precisely, this means that beside the requirement to own a valid DDI, much consideration must be put into access times and limits for the public functions that users use to interact with the system.
Besides the core functionalities of a cryptocurrency, the system issues coins to reward users for strengthening the system by participating. The block creation is a core task in the system. Without the creation and consensus about the next block, the system does not transit its state. In Bitcoin, the selection of the next block depends on the solution of a difficult problem. The hash of the block (which is interpreted as a number in that case) must be below a threshold. Bitcoin pursuits a constant block creation time. The threshold is adjusted by comparing the desired block time with the real block time over a predefined amount of time that has passed. If the block time is less than desired, the threshold is adjusted such that it is more difficult to find a block with a hash that fulfills the condition. If the block time is higher than desired, the threshold is adjusted such that it is easier to find a valid block[Nak09]. This process is called "Proof-of-Work" (PoW). It couples the validity of a block to power (electricity) consumption. The creation of a valid block is rewarded with newly generated coins and transactions fees. An alternative to this block creation process was introduced in 2012 and is called "Proof-

of-Stake" (PoS)[KN12]. Instead of solving a difficult problem, the intrinsic currency is staked (locked) and the staker receives the right to create blocks in proportion to the total stake. The stake is slashed, if the associated user acted maliciously. This is the rudimentary idea, many variants of PoS exists to handle some problems PoS in its pure from brings with it[BMZ18]. It was developed because the amount of electricity consumed for Bitcoin PoW was unreasonably vast. Although it reduced the electricity requirement to a minimum and introduced a semi-democratic process of choosing the next block, it also comes with some drawbacks. First, the higher the stake a user can supply the higher the influence the user can wield. This shares some similarities with PoW. The more they can spend on hardware, the more frequent they will find valid blocks. Another potential drawback is that a decline in price can lead to a snowballing effect in the system. Profit-orientated stakers could (temporarily) withdraw their stake (after a lock period has passed) and sell it on the market. A lesser total stake means that a lesser stake is required to be able to act maliciously.
During the first phase of the system suggested in this work, "Nominated Proof-of-Stake" (NPoS)[CS20], a variant of "Delegated Proof-of-Stake" (DPoS), is used, where for a limited period of time a committee is elected by the community to validate blocks. The algorithm ensures that a overrepresentation of a minority does not occur. Using a PoS variant initially leads to the question how should users stake, if there is nothing to stake in the first place? If the system has not issued any coins yet, nothing can be staked and therefore the first block after genesis cannot be authored. The following solutions tackle this problem:

1. Issue coins to users before the system launches. For example to those users, who supported the system during its test phases before the main network was released
2. Sell coins to users before the system launches.
3. Use the Polkadot network and allow users to stake with other cryptocurrencies from the Polkadot network (inherits security)

After the system ran some time and enough DDIs have been created, NPoS can be switched to a new yet unimplemented block authoring version: Proof-of-Identity (PoI) based on DDIs that are each bound to exactly one unique human being. Any DDI can vote for a block in return for coins. The stake in this scenario is the right for a specific human being to participate in block authoring. If a DDI acts maliciously, it is locked out from block authoring for a specific period of time. This block authoring mechanism has a major advantage: It is truly democratic and it is economically reasonable (in terms of resources). Due to time restrictions this idea is not further investigated in this work, but future research is highly encouraged due to the previously mentioned advantages.

A tiny fraction of coins is generated for block authoring, in early stages by . In later stages, voting based on a PoI protocol is rewarded. The major fraction of coins is generated as rewards for helping the system to evolve and operate:

1. Proposing ideas and concerns and voting for them
2. Serving in a community-elected council to resolve conflicts and difficult decisions
3. Working in a project
4. Supplying storage space
5. Supplying computation time (to run projects for example)

3.2. Blockchain modules (FRAME-pallets)

In this section an overview of the FRAME-pallets that are used in this system is presented and their function and role is elucidated. In addition, the interaction of the custom pallets, the pallets that were designed and (partially) implemented in this work, is clarified.

3.2.1. Overview

Figure 3.2 depicts all the "Framework for Runtime Aggregation of Modularized Entities" (FRAME) pallets that are required to run the system in its first version. FRAME pallets are modules of the Substrate blockchain. They can provide functionality that is either invoked by the Substrate core, other pallets or via a call that was submitted within a transaction. Here they are grouped in 3 categories: "Fundamental", "PoS" and "Custom". Fundamental pallets and PoS pallets are provided as default pallets from Paritytech (the organization behind Substrate and Polkadot). The function and usage of them is documented in the "Application Programming Interface" (API) documentation[1]. Custom pallets encompass the functionality of the system proposed in this work and have to be implemented from scratch, whereupon the functionality the fundamental pallets provide is used. PoS pallets are used to provide PoS functionality and are not directly invoked by the custom pallets.
The frame-system pallet is required by any other pallet. According to the documentation:

> The System module provides low-level access to core types and cross-cutting utilities. It acts as the base layer for other pallets to interact with the Substrate framework components.

[1] `https://substrate.dev/rustdocs/v2.0.0/frame_system/index.html`

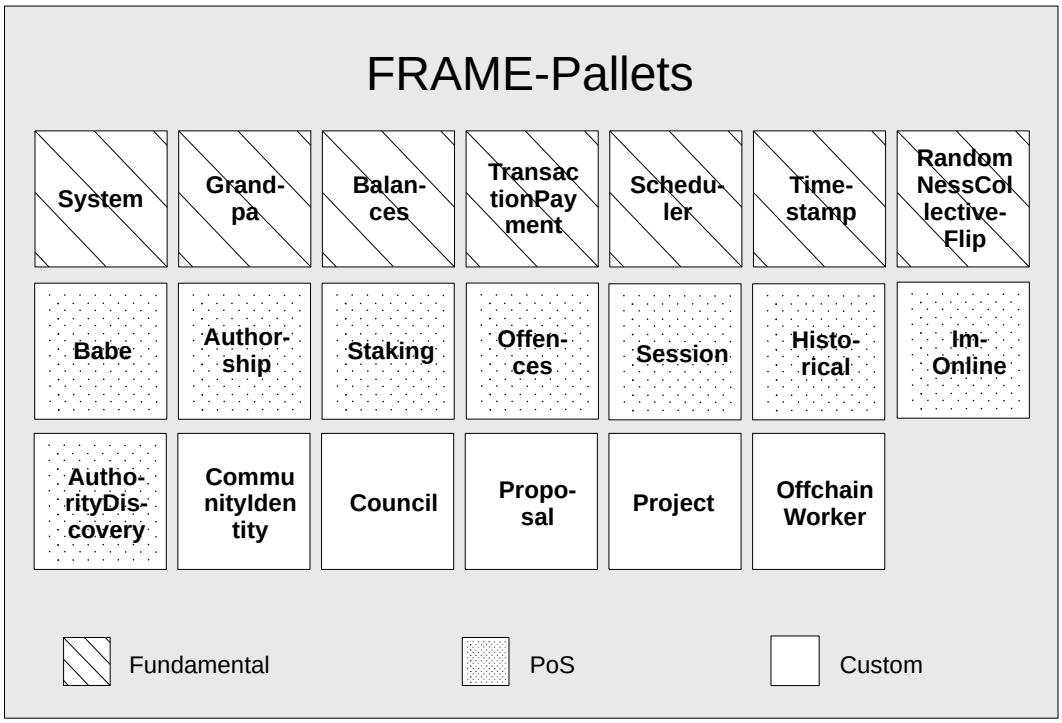

Figure 3.2.: FRAME-pallets required for this system

The "GHOST-based Recursive Ancestor Deriving Prefix Agreement" (GRANDPA)[SK20] is a finality gadget that provides provable deterministic finality to a block. When using PoS without a finality gadget, finality is probabilistic. That means that any chain that becomes the longest chain will replace all other candidates, without any security that a block won't be discarded. The older the block, id est the more blocks are between it and the current tip of the blockchain, the more unlikely it is that it will be replaced. This is because multiple validators must consequently vote on new blocks that use that old block as their ancestor. Therefore finality is probabilistic and never 100% guaranteed. Depending on the consensus algorithm, other criteria can be used to select the current blockchain. For example, in Bitcoin (which currently uses PoW) instead of length, the total difficulty is used as a measurement. GRANDPA provides a secure mechanism that results in deterministic finality of past blocks. This is achieved when a majority of 2/3 of voters have voted for a block to be final. The finality of blocks between a final block and the tip of the blockchain is still probabilistic. Experiments with the public Polkadot blockchain have shown, that deterministic finality is most of the time reached for the third block behind the tip (fourth most recent block). By splitting the finality gadget from the block production and chain selection rule (like NPoS), freedom in the selection of combinations of those three procedures is given and a formal proof of GRANDPA is possible.

The Balances pallet provides control over the creation and destruction of coins and a versatile management of account balances. The simplest form of balance is free balance. It can be freely used by the owner. Balance can also be reserved, in that case it is unusable. It can also be locked until a specific block number. The pallet also provides imbalances, a mechanism to keep track of coins that have been issued or deducted or increased in an account without and equal increase or deduction in another account.
The TransactionPayment pallet provides logic that determines the minimum fee required for a transaction to be included. It is consisting of a fee proportional to the length of the transaction, a fee proportional to the computational and storage requirement (weight) for the execution of pallet invocations and an optional tip.
The Scheduler pallet adds the functionality to schedule and cancel named and anonymous tasks for a future block number. When the block with the proper block number is initialized, the pallet includes a call of a pallet function into that block (or a block shorty after it, depending on the schedule priority and blockchain load).
The timestamp pallet adds functionality for block validators to add a timestamp to a block. Other validators reject timestamps that are too far off their local clocks. The blockchain measures time in block numbers. The lowest amount of time that can pass in the blockchain is one block. The system tries to be as close to the desired block time as possible by using weights to calculate the execution time on a reference CPU. The real block time is almost always close to the desired block time, but over time the small difference of real and desired block time accumulate and therefore it is difficult to create appointments, for example. In scenarios that require the blockchain to act according to a timestamp, the timestamp module should be used.
The RandomnessCollectiveFlip pallet is used to retrieve a low-influence random number, whose computation encompasses the inclusion of the block hashes of the previous 81 blocks. They should only be used in situations that have a low security requirement. The sole purpose of this pallet in this project is to deliver a seed for a random number generator that is used during block production.
The previously described pallets are the fundamental pallets, the next pallets belong to the PoS group.

The “Blind Assignment for Blockchain Extension” (BABE) protocol[Alp20] allows users to assign a locked stake to a “Verifiable Random Function” (VRF)[MRV99] key. This key is used to produce secure randomness, which in return is used to decide if the user is allowed to produce a block. The author provides a solution to avoid grinding of keys and interrelated predetermination of block production opportunities.
The Authorship pallet does log the current author and the past uncles (for a configurable count of past blocks). It is used to reward block authors and authors of uncle (valid but omitted) blocks. When a longest chain does replace another chain as the

canonical chain, the reward that was previously paid out to block authors is not included in the canonical chain, if their block candidate is not part of the new canonical chain anymore. Nevertheless, they have contributed a valid block which is now an uncle and should still be rewarded (less than for an included block) for it.
The Staking pallet offers an interface for users to bond a stash account and a controller account to directly or indirectly participate in block authoring. The funds in the stash account are frozen, the controller account participates in block production, knowing that malicious behavior will eventually lead to the complete or partial slashing of the stash account. The controller account can either be a validator or nominator. A controller account that announced interest in being a nominator, is assigned this role in the next election round. To become a validator, a controller account registers as a nominee. Nominators will vote on nominees during the next era, whereupon the voting power scales proportionally with their stake. The elected validators must be online at any time and participate in block authoring when its their turn. They can either validate blocks or ensure their finality, in this case by use of the BABE and GRANDPA pallet respectively. Nominators are encouraged to vote for obliging by the circumstance that their rewards or punishments are coupled to the rewards and punishments of the validator they voted for. In summary, while the BABE and GRANDPA pallet are used for block authoring, the Staking pallet is used to elect validators and authorities who participate in block authoring during an era, and to reward or punish all involved parties according to the outcome.
The Offences pallet is loosely coupled to the Staking pallet by a trait (interface) definition (it can be swapped out by another pallet that implements the trait). Users are encouraged to observe the blockchain and report malicious behavior to the Offences pallet in return for a potential reward.
The Session pallet handles the rotation of sessions and provides functionality for the management of session keys. A session is a period of time during which the validator set is fixed. The session keys are required by the validators and authorities to fulfill their tasks.
The History pallet is a module within the Session pallet, but for the sake of clarity it is treated as an separated pallet. A usual approach in NPoS is to lock the funds of stakers (nominators and validators) longer than they keep those roles. This ensures that users of the system have more time to report malicious behavior. Furthermore, it protects the system from arbitrary cash outs by stakers in case the currency temporarily loses value on the market, and therefore partially protects from a snowball effect where the system rapidly loses security due to massive reduction of staking value. Since the unlocking of the stake is only delayed, such a scenario is still possible. The history pallet does include the merkle root of a session to the blockchain, which is reasonable because it has only the length of the digest size of the hash that was used to generate the trie.

Those merkle tries contain session data from validators during their incumbency as their leaves. If malicious behavior is found, its existence can be proven by a merkle proof from the associated session data. This modules in summary provides a way to efficiently (in terms of storage and computation requirements) prove that a validator behaved maliciously in the past.
ImOnline is a pallet providing validators with the ability to signal that they are online during a session. This is required because validators should not be selected for the authoring of the next block when they are not available. Keeping track of those signals during one era can be used to punish validators for not fulfilling their duty.
AuthorityDiscovery is a very rudimentary pallet that is used within the Substrate core networking module to check whether incoming connections play a role in block creation or finalization.
With AuthorityDiscovery, the pallet of the PoS are complete. The following pallets belong to the Custom group and are specified and partially implemented from scratch.

The CommunityIdentity pallet is at the core of the whole system. It is the base on which almost any functionality is build on. It adds an authentication layer that is based on DDIs and ensures, that every DDIs is mapped to exactly one real physical human being. This allows the whole system to operate completely democratic, with an equal power distribution in regards to voting. It provides the users the possibility to request the verification of some physical traits of their identity. The verification can have different false-positive probabilities. Depending on the requirements of a specific democratic procedure and the probability of the identity being a false-positive, they can gain the privilege to participate. With nothing but the fact that some of their physical traits have been verified, they are granted the privilege to put a hand on the ruder of the morphogenetic blockchain. They in some sense, get a part of it and its up to them to influence the direction of the evolution. The pallet provides functionality to map an account address to the verified identity. By submitting transactions that are signed by the private key that underlies the address, any module within the blockchain is able to check the associated DDI of that address by using this pallet.
The council is required to resolve conflicts and solve difficult decisions as a representative of the community. The Council pallet realizes this concept by providing the following functionality:

1. Periodic election of the council
2. Delegation of decisions to the council, which are solved by voting
3. Rewarding for participation and correct votes
4. Forcing the removal of inactive council members

5. Offering the possibility to force a reelection by a supermajority vote of the (identified) community

The last two features encourage the council to not act on their own behalf and therefore to embody a servant of the community.
Proposal serves the community as an interface to share their ideas and concerns, with the potential outcome of them being realized by the community and paid by the system. It handles the management of proposal and concerns, their associated votes, a final vote by the council and ultimately the conversion into a project.
The Project pallet does provide features that allows a decentralized community-funded realization of a proposal. It support the buildup of a team, payments to its members, conflict resolution and the inspection and approval of the hand over. When a project is finished, it becomes a public good and is accessible by anyone. Projects that require a shared database or complex computations can be run by the community in exchange for rewards, with is handled by the last custom pallet, OffchainWorker.
The OffchainWorker pallet primarily exists as a connector between the IPFS storage and the Substrate blockchain. Since it is technically unreasonable to store arbitrary data on the blockchain, the data is stored in IPFS instead. A "Content IDentifier" (CID) is returned, which can either be a hash of the data alone or some additional metadata. The CID is stored on the blockchain. By using an appropriate front-end, the user will not notice that the data is not stored on the blockchain, because the front-end resolves the CID to the associated data. Storage providers can register on the blockchain. In return for pinning data (keeping it completely in their storage), they are rewarded by freshly generated coins. The pallet provides functionality to measure, whether a storage provider pins the data. It also provides functionality to increase their reward based on their total available bandwidth. Another function of the pallet is to reward users that supply computation time. Some applications that were developed by the system proposed in this work might require expensive or specialized hardware that not every user has at hand, to be executed in a reasonable time. Computation time providers can add those application to their portfolio. This mechanism does lead to the accomplishment, that the system does not only facilitate the implementation of proposal by the community, but also does facilitate rewards for infrastructure required to operate the resulting applications.

3.2.2. Relationship of the custom pallets

Figure 3.3 shows the relationship between the custom pallets. An arrow indicates, that the pallet the arrow starts at depends on the pallet the arrow points at. The OffchainWorker pallet is used by all custom pallets. Because arbitrary storage of data is disencouraged, only CIDs that refer to the data in the IPFS storage are stored in the

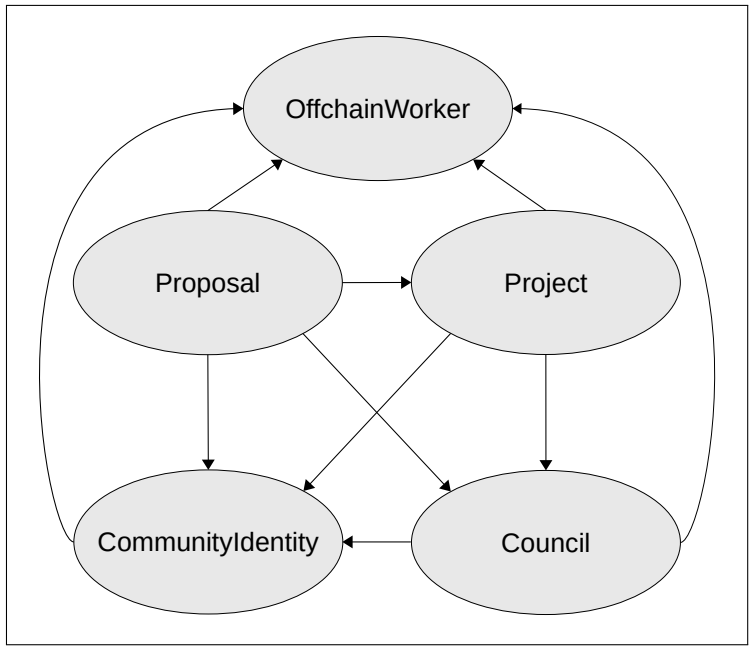

Figure 3.3.: Relationship of the custom FRAME pallets

pallets. The pallets register those CIDs in the OffchainWorker pallet. Storage providers can query those CIDs and can register themselves as providers of the associated data. The initial storage of files before the CID is announced in the OffchainWorker pallet marks a special case.

The Proposal pallet depends on every custom pallet. It uses the CommunityIdentity pallet to ensure that every DDI stays within its proposal, concern and vote limits. It uses the Council pallet to register a vote for or against the spawn of a project from a winning proposal and its associated concerns. When the council decides for the proposal, the Project pallet is invoked to create a new project and transfer the proposals and concerns into it.

The Project pallet depends on any pallet but the Proposal pallet. It ensures that every participant in the project is unique by using the CommunityIdentity pallet to convert an address into a DDI. If a project team acts maliciously, it can be reported. If enough reports are collected, the project pallet invokes the Council pallet to start a poll. The council has to decide whether to abrogate the current project team. After some time, the project pallet fetches the result and handles it adequately.

The Council pallet depends on the CommunityIdentity and OffchainWorker pallet. The CommunityIdentity pallet is invoked to get the information about a potential candidate for the council. It used to check if the user owns a DDI and if the probability of the user having a false-positive identity is low enough.

The CommunityIdentity pallet does only use the OffchainWorker pallet to store a set physical traits, such that future users that apply for verification can not have the same

physical traits.

3.3. Identity pallet

This section offers a conceptual view on the identity pallet. Section 3.3.1 describes the physical traits and algorithms that are used to establish a one-to-one mapping between an human being and a digital identity. Section 3.3.3 describes the decentralized validation scheme of the digital identity, which effectively turns it into a DDI. In section 3.3.4 a mechanism is presented to estimate the false-positive probability of a DDI containing not the physical traits of the associated human being during creation. Section 3.3.5 lists some dangers of this mechanism that should be regarded during further development of this identification scheme. This section is closed with a suggestion that targets how the initial identities are validated in section 3.3.6.

3.3.1. Basis of the identity

The basis of the DDI is biometric data. Since different physical traits of the human body offer different qualities as biometric data[1], a distinction of the most suitable traits for this system is of great importance. Major qualities of the underlying physical traits are: Security, accuracy, permanency and usability. The security quality measures how difficult it is for attackers to deceive the system and pretend that they own the traits of another person. Accuracy describes, how accurately the traits were measured. Often it is measured by how likely it is a system interprets two different traits as equal, which is called the "False Acceptance Rate" (FAR), or interprets two equal traits as different, which is called "False Rejection Rate" (FRR). Permanency describes how fast the traits change and usability describes how complex the circumstances are under which a correct measurement can be achieved.
Since no 100% accurate algorithm exists, human interaction is required to ensure that false-positives are sorted out, id est that a identification request is not rejected due to a false match. Demanding optimal hardware and conditions is impractical in an permissionless decentralized system, since it would exclude a majority of users from participating. Nevertheless, users that request an identity must be instructed to establish the best possible conditions regarding their circumstances. In terms of security, the validators must take action to test whether the audited users show their real traits. Support by algorithms that are able to detect fakes, for example those generated by a computer, can lead to better results. The traits should be scannable using non specialized hardware. The iris and the veins for example are therefore inapplicable. The structures in the iris are very fine granular and a dark iris requires more light. A

[1] `https://towardsdatascience.com/biometric-authentication-methods-61c96666883a`

usuable scan of the veins requires infrared cameras. On the other hand, traits like the voice can be distorted by the interviewee. Traits that can be distorted easily are the most undesirable ones, because the interviewees are not trying to authenticate, they are trying to obtain an unique identity that is bound to an address and therefore a private key they own, that they will use to authenticate once their identity was validated. Consequently, a system that checks such traits scanned from an interviewee against those already stored can easily be deceived, which ultimately allows the interviewee to create infinite unique identities.
Two traits are appealing for the aforementioned constraints: Fingerprints and facial traits. Fingerprints have the advantage that the storage size is small. In addition, during the evaluation of this method during this work it turned out, that with just display brightness and a camera having only 1.3 megapixels it is possible to clearly see the unique fingerprint. Further experimentation is required to verify that this information is enough for an algorithm to reliably extract the features. One downside is that many people feel uncomfortable to store their unique fingerprint in a system and therefore the acceptance might be too low. Another downside is that auditors are hardly able to verify whether a result is a false-positive (they have to compare the fingerprints by themselves). Pictures are less of a problem in terms of privacy, usually people are more willing to publish a picture of them in comparison to fingerprint. Matching results are also much easier revealed as a false-positive by human auditors, because evolutionary it was more important to differentiate facial properties in contrast to characteristics of a fingerprint and therefore the brain is more specialized on this task. A slight drawback is the template size, the size of the features (and metadata) extracted from a picture to be eventually matched. As it is shown later in this section, facial feature matching algorithms have become extremely effective. Due to the fact that auditors have to reappraise matches, the facial traits will be used as the physical traits that map a DDI and a unique human being.

3.3.2. Selection and evaluation of the facial recognition algorithm

P. Grother et al. from the “National Institute of Standards and Technology” (NIST) performed an extensive benchmark of facial recognition algorithms[GNH20]. They examined 239 proprietary algorithms in respect to many different properties, in case the algorithm supported it. The underlying database consists of about 12.3 million mugshots with a resolution of 480x600 pixles and a filesize in the range of 18KB to 36KB. In future reviews, they plan not to restrict the database to mugshots only. Extracting the features of each picture by a facial recognition algorithm into a database is called enrollment. Three different strategies of enrollment are evaluated: Recent,

lifetime-consolidated and lifetime-unconsolidated. Enrolling only the most recent picture of an human being is called recent strategy. Enrolling a sequence of pictures within an arbitrary time span and excluding the most recent picture of the same human being, whereupon the facial recognition algorithm is supplied with the information that those are pictures of the same human being, is called lifetime-consolidated strategy. The lifetime-unconsolidated strategy is identical to the lifetime-consolidated strategy, with the exception that the algorithm does not know that they depict the same human being. Depending on the application, one strategy can be more suitable than another (refer to the document for more information).
The quality of an algorithm in regards to matching accuracy is measured in "False Positive Identification Rate" (FPIR) and "False Negative Identification Rate" (FNIR). Note that more methods exists to measure other qualities of the algorithm, but since they are not further relevant in this work, they are not considered.

$$FPIR(N,T) = \frac{\texttt{Num. non-mate searches with threshold T that yielded a result}}{\texttt{Total non-mate searches}} \tag{3.1}$$

whereas N is the total count of enrollments and T is the scoring threshold that is used to decide whether to drop a result or to keep it. In words, this identifier shows the fraction of searches with a score threshold of T in a database with N enrollments, that contain at least one positive match although the human behind the input picture was not enrolled before (id est no template for this person exists).

$$FNIR(N,R,T) = \frac{\texttt{Num. mate searches where enrolled mates rank>R or score<T}}{\texttt{Total mate searches}} \tag{3.2}$$

whereas N is the total count of enrollments, R is the number of top-scoring matches that are returned and T is the scoring threshold that is used to decide whether to drop a result or to keep it. In words, this number shows the fraction of how many searches with a score threshold of T in a database with N enrollments, do not return the appropriate template within the top R scoring matches, although the corresponding template exists.
The FPIR identifier is especially useful for systems that authenticate or identify, for example where a decision is made automatically (automatic authentication at a door for example) or where a wrong identification could have severe impact for the subject (surveillance). By raising the threshold, the FPIR lowers and the FNIR rises. For automated systems, this is desirable. A door that has a higher chance to fail at matching an authorized face in exchange for a lower chance to match a face that is not authorized is highly preferable. In the system proposed in this work, authentication happens by emitting a signature based on a private key that is bond to an identity. In addition, this system does offer a review process during which enough auditors exist to examine

the results and spot false-positives (refer to section 3.3.3). Consequently, the situation is inverse: A higher chance to falsely match the face of interviewees (who requested an identity) in exchange for a lower chance to fail at matching the interviewees faces and therefore not recognizing that they already own an identity, is highly preferable. Therefore the FNIR of an algorithm should be as low as possible. Since a higher threshold results in a higher FNIR, the threshold is set to 0. This has side effect: Since for any search request the algorithm returns at least one result ($T = 0$), FPIR is constantly at 1. The other parameter R must also be set such that it is balanced between the negative impact on the FNIR (for lower values) and the higher effort for the auditors (for higher values) to review the false-positive results. This is due to R constraining how many matches are returned by the algorithm at most. If R is set too conservative, for example only the best match ($R = 1$) is emitted, the chance to miss the correct template in the search is higher than if multiple matches are returned and investigated. If R is set too unstinting, the auditors have to manually match a unreasonable large number of up to R results against the interviewee. One of the best algorithms in terms of low FNIR and search duration, is labeled as "NEC-3". Table 19 in the document that evaluates the facial recognition algorithms compares the FNIR for different rank-limits R, with a fixed score threshold of $T = 0$ and an enrolled dataset of 1.6 Million mugshots. In this case the "recent" enrollment strategy column is relevant. The table shows that for one of the top algorithms NEC-3, that when raising R from 1 to 10, the NPIR is reduced from 0.0014 to 0.0012, which is about 14.3%. A raise of R from 10 to 50 doesn't change the NPIR any further, as stated in the table. Since rounding was applied, a maximum possible change of 8.33% is still possible (assuming rounding to nearest integer was used as the strategy), but might not be reflected in the table. Nevertheless, a reduction by 14.3% in NPIR by increasing R from 1 to 10 in comparison to a maximum potential change of 8.33% by increasing R from 10 to 50, shows that the NPIR and the R have a non-linear relationship, the decrease of FNIR per emitted match is less the higher the value of R is. Therefore an increase from 1 to 10 results in a higher gain in contrast to an increase from 10 to 50 or 50 to 1000. This indicates that the algorithm already matches very precisely, because the correct match is most of the time found in the top ranks. In conclusion, $R = 10$ seems like a adequate selection, especially because the auditors are easily able to compare 10 possible matches to the interviewee in an acceptable time span.

How long does the algorithm need to match and return R pictures in dependence of the enrollment count N? This question should not be neglected, because after all, human beings are waiting for the results to compare them. Furthermore, what is the acceptable maximum? This depends on the application and the subjects. For example, standing in front of a door and waiting for it to match the own face for more than a couple of minutes should be unacceptable for most, if not all people. In this work, the

auditors have to wait for the matching results during the interview (identification process). As a matter of principle, the proposed system in this work should be unlimited and therefore be able to match potentially billions of people in a reasonable time. If this system should ever reach such an enormous size, decades most likely have passed and technology has made major advancements. The benchmark report does currently use datasets with 1.6M, 3M, 6M and 12M pictures. This should be sufficient for the near future. Table 4 in the benchmark document shows search times for varying R and N values. The benchmark was executed on a single core of the "IntelR©XeonR©CPU E5-2630 v4 @ 2.20GHz" CPU. The first insight is that changing R does merely impact the search duration for most of the algorithms and in the most cases can be discarded as being within the margin of error. Different N values, id est a different amount of enrollments, does impact the search duration. For most of the algorithms, N and the search duration have a linear relationship for databases with more than 1.6M enrollments. The search duration ranges from less than a millisecond up to 15.4 seconds for a database with 1.6M enrollments. NEC-3 needs about 7 milliseconds at 1.6M enrollments, at 12M enrollments it needs 82ms. It should be easily able to match the systems population against a picture in under a second within the next decade. Assuming that the relationship is indeed linear based on the four data samples, it should be able to find matches in about 8 seconds for a database containing 12 Billion enrollments on one single core of an contemporary CPU. NEC-2 has a lower FNIR, but at 12M enrollments already needs about 4.3s to match a picture.

The template generation time and the template size are two further factors that have to be respected in the selection of an algorithm. For every identified user, two datasets exist. One is the template, which the algorithm uses to match pictures against. The other dataset contains the pictures that is associated with the template, so that the auditors can compare false-positives manually with the interviewees appearance. According to table 4 in the benchmark document, the NIC-3 algorithm needs 1712 Bytes per template. Assuming that pictures are saved in the same quality as the enrollment pictures in the benchmark, the total size required per DDI does add up to $1,712Bytes + (18,000Bytes + 36,000Bytes)/2 \approx 27.700Bytes$, assuming that the pictures are evenly distributed between 18KB and 36KB size. For 10 million users, already 270GB are required. This is still realistic and even already manageable by casual computers. On a single core of the "IntelR©XeonR©CPU E5-2630 v4 @ 2.20GHz" CPU, the template generation takes 690ms. This is acceptable. In this respect, the state of the art of facial recognition algorithms does offer acceptable template generation time and sizes, even when storing the image that is associated to the template.

Since the system should be able to handle a high number of DDIs, the relation of the FNIR to the amount enrollments (effectively the number of DDIs) is of interest. Fortunately this benchmark was also executed by NIST and is documented in the benchmark

document in table 16. During the evaluation T was set to 10 and R was set to 50, but as it was already shown in this section, R should not have a noteworthy impact on FNIR when NEC-3 is used. Astonishingly the result for NIC-3 is that the FNIR is constant at 1.6M, 3M, 6M and 12M enrollment templates. Unfortunately databases with a lower amount of enrollment templates were not tested. Intuitively the more datasets exists, the higher the probability to match a picture incorrectly. This assumption is confirmed by the benchmark results of other facial recognition candidates in table 16. At the current FNIR, assuming the system has 10 million DDIs and all of them try to get a second DDIs, 10 million reviews would be scheduled. During this meetings, every person is matched against the enrollments. When using an algorithm that is in the same class as NEC-3, the FNIR is 0.0012. Out of 10 million DDIs, 0.12%, which are 12000 DDIs in this case, succeed in creating duplicates. In the next sections, a counteractive measure is presented to punish duplicate DDIs requests, which should lower the number. If this measure is effective, the number of users who are willing to take the risk are reduced greatly. In addition, history shows that facial recognition algorithms improvement accelerated quickly in the past years, as stated by NIST:

> The major result in NIST IRs 8238 and 8271 was that massive gains in accuracy have been achieved in the years 2013 to 2018 and these far exceed improvements made in the prior period, 2010 to 2013 [...] This is evidence that face recognition development continues apace, and that FRVT reports are but a snapshot of contemporary capability.

Assuming this trends continues, those algorithms will further reduce their execution time, storage requirements and FNIR and FPIR values (as well as other qualities not regarded in this work).
Table 16 in the benchmark document shows benchmarks of the algorithms FNIR in relation to a FPIR with $R = 1$ when matching quality mugshots against previously enrolled quality mugshots. This is not exactly the same scenario as described in this work, because webcam images are matched against the best possible images a user could supply. Figure 106 in the benchmark document shows the benchmark result of the FNIR in relation to the FPIR when matching webcam pictures against a database of enrolled mugshots. Note that the setting is different than it is in this work. The subject pose was not constrained and therefore the algorithms also had to match pictures of profile views, partial faces, multiple faces in the pictures and some more scenarios that will not occur in this work due to supervision. The FNIR value that is interesting in this case is the FNIR value for a fixed FPIR value of 1, consequently $T = 0$. In this case, NEC-3 has a FNIR between 0.005 and 0.009 (it is impossible to extract the exact value out of the figure). The fact that auditors should instruct interviewees to take an optimal pose during a DDI creation process does further reduce the FNIR.

Based on the benchmarks, NIST states the following:

> With good quality portrait photos, the most accurate algorithms will find matching entries, when present, in galleries containing 12 million individuals, with rank one miss rates of approaching 0.1%. The remaining errors are in large part attributable to long-run ageing, facial injury and poor image quality

In the system proposed in this work, auditors ensure that the pose in the image is optimal, therefore poor image quality in the sense NIST meant it in the quote, is eliminated. To prevent aging and facial injuries to lead to multiple DDIs for the same human being, the system does enforce a refresh of the picture and the associated template stored in the database. This should in practice lead to significantly better results than those that were benchmarked.
The enrolled mugshots have a resolution of 480x600 pixels. By instructing interviewees to come close to the camera, about 50 percent of the picture are filled with the interviewees face. Even with a 1MP camera, the face already occupies around 500x500 pixels. Instructions that lead to optimal lightning conditions in relation to the situation the interviewee is in and the enforcement of a neutral facial expression can even further increase the enrollment quality. Modern technology that uses artificial intelligence can be used to increase the picture quality. By taking a short video sample of the interviewee, during which small movement is executed, a tool like TecoGAN[Chu+20] can add information to the video and the picture quality can be increased. This solution enables low quality webcams to be used to take higher quality pictures that are used for enrollment and matching. In addition, shadows in the final result can be removed in most cases by using artificial intelligence, just like it is proposed in the "Portrait Shadow Manipulation" paper[Zha+20].
To summarize this section, it showed that current state of the art facial recognition algorithms are able to support this systems in regards to execution times, storage requirements and accuracy. Naturally, those algorithms must not be used in the system proposed in this work due to their proprietary nature. Using them would violate the requirement, that the system must be trustless. Nevertheless, since the extensive benchmark and analysis has shown that contemporary algorithms can handle the technical requirements, it is only a matter of time before open source solutions are crafted which then can be used in the system. The section was closed with suggestions on how to minimize the FNIR although low quality webcams are potentially used for capturing pictures.

3.3.3. Decentralized validation scheme

Registration and selection of auditors

Figure 3.4 shows the registration process for a DDI audit of parts of the identity, in

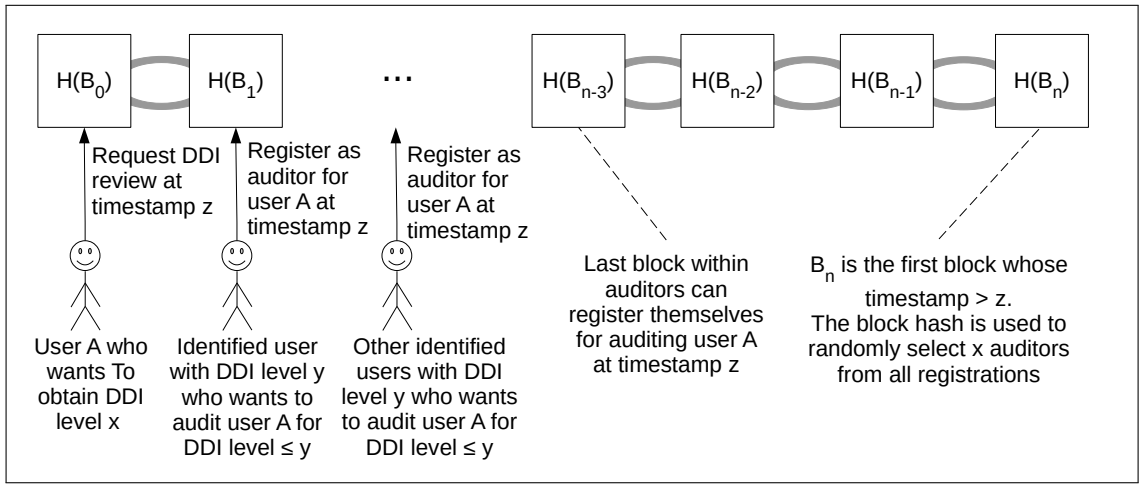

Figure 3.4.: Overview of the DDI audit registration process

this case the facial traits, of an human being. User A wants to obtain a DDI with identity (verification) level x. The purpose of the identity level is described in section 3.3.4. The user adds a timestamp of the moment in future when the audit should happen. The registration process requires the users to reserve a specific amount of coins from the system-intrinsic currency. If the verification process succeeds, the reservation is removed and the user gains control over those coins again. If the verification process fails, the coins are burned and consequently the users loses those coins. This mechanism is a protection against "Denial of Service" (DoS) attacks. If no punishment for malicious behavior exists, users can request vast amounts of DDI audits by only paying transactions fees. This stalls the DDI audit process for honest user, because the limited number of auditors will be exhausted for a long period of time interviewing the malicious users. After user A registered a timestamp as an interviewee for the audit of DDI level x and the funds (whose amount depends on the request DDI level) were reserved, users with a DDI level of x can register to audit user A at timestamp z. This is possible until $(z - \texttt{timestamp\_now}) \cdot 3 \cdot 1.5 < \texttt{blocktime} \cdot 3$. `timestamp_now` is the current timestamp from the perspective of the current block and `block time` is the desired block time. The number 3 is used to find the 4th last block before the auditors are randomly selected, this is block B_{n-3} in figure 3.4. The multiplier 1.5 is used to adjust the calculation such that the real block time is considered instead of the desired block time. The block time is neither constant nor deterministic. By the help of extrinsic weights it is possible to estimate the execution time for an extrinsic, but in reality the real block time is slightly higher most of the time. By observing the

main Polkadot network it is possible to estimate the difference. After all the difference depends on how accurate the weighing functions are in the pallets. The real block time can get up to 30% higher (this happens rarely). Usually it is less than 10% higher. If the calculation of the block number on which the registration is closed does not respect this, the event that the registration is closed after the desired audit timestamp becomes likely. When `timestamp_now` $> z$, then the pallet uses the previous block hash as the input for a linear congruential generator with at least 32 bit output to randomly select x. The reason it uses the previous block hash is because at the time of execution, the current block hash is not determined yet. Closing the registration and the random selection of the auditors happen some blocks apart, because this approach makes it impossible to know the block hash in advance and therefore potentially influence ones chances by calculating the result of the linear congruential generator for any casual user. Block validators have a greater advantage, because they compose the block and therefore have full control over its contents (as long as it obeys the consensus rules). By including different transactions or ordering them differently within a block for example, the resulting block hash changes. A block author can therefore try different block compositions and calculate the set of auditors by using the block hash as a parameter for the linear congruential generator. This is unavoidable. Since the NPoS algorithm ensures that a minority of block authors is not overrepresented, block authors with high stakes do not get the chance to author a disproportionately high amount of blocks and consequently cannot use their wealth to influence the DDI audit process significantly. A mechanism that rewards users for pointing out malicious behavior of block authors or the implementation of a mechanism to add a random number that was calculated in a distributed manner by block authors is highly recommended, but is not further examined in this work.

Establishing a meeting

The meeting is held by using an off-chain software. The software keeps a local enrollment database, if the operator participates as an auditor, that must be kept up to date. The identity pallet does keep track of the CIDs of approved DDIs associated pictures, that the software can use to fetch missing DDIs from IPFS. The participants do establish a peer-to-peer connection. When a new incoming connection is opened, a challenge is sent out. This challenge can simply consist of a unique name for the tool and the current timestamp. The challenger does sign the challenge and return it. In case the public key associated with the signatures (indirectly through the private key) does match to an address of the interviewee or a previously selected auditor (which can be queried from the pallet), the connection is kept alive. By using this approach, it is guaranteed that nobody can enter the interview but the interviewee and the selected auditors. If every participant is connected, the video sessions start and the auditors

start to verify that the identity does not already exist.

Executing the audit

Figure 3.5 shows the different stages of the audit. During the first stage, the parti-

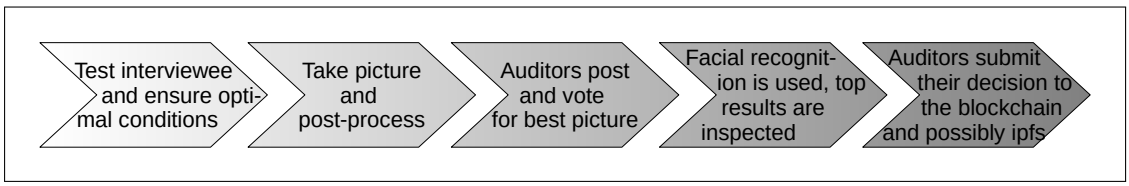

Figure 3.5.: Flow of the audit process

cipants introduce themselves and the auditors ensure that the conditions are optimal. They do so by telling the interviewee to find the optimal brightness, direction of light if possible and pose. After that they verify whether the interviewee is real, id est does not wear any masks or uses computer generated images. They can do so by instructing the interviewee to move a hand in front of the face with open fingers, turn the head slowly, ask questions that a computer is unlikely to answer reasonably, etc. A guideline should be developed for this part of the process. If the auditors are convinced that the person they interview is real, they can enter the next stage.

During that stage, the auditors instruct the interviewee to keep a neutral face. They take some pictures, if necessary use artificial intelligence to increase the image quality for example with TecoGAN[Chu+20], remove shadows[Zha+20] and possibly apply additional post-processing which does not change the features of the face. If everybody is content with one picture of the interviewee, the picture candidate is shared with all other auditors.

In the next stage, the auditors have a limited time frame to vote for their favorite. The picture with the most votes wins. If no picture has the most votes, the top performing candidates in terms of equal votes compete. The software of every auditor selects a random 256 bit number and posts the hash (the tools implementation dictates the hash function) of it. After that the software of every auditor reveals the preimage, everyone is able to verify that the number was predetermined and not changed after receiving the numbers from the peers by hashing it and comparing it to the hash that was commited by the corresponding peer. It is important that the numbers are randomly selected, so a rainbow table has no effect. At last the numbers are concatenated, hashed and the resulting hash value is taken modulo the number of the highest voted pictures with equal votes. By using this approach, the peers can truly randomly choose a winning picture in case of a tie.

In the next stage, every auditor uses a predetermined adequate facial recognition algorithm, for example an open source algorithm that performs as efficient and effective as

NEC-3, and inspects the R top matching results ($R = 10$ for an algorithm like NEC-3). If a correct match occurs, the auditor must reject the DDI. In case of acceptance, the picture that was matched against the enrollments must be enrolled, such that future audits incorporate the new user. In that case the OffchainWorker pallet is used to submit the enrolled picture to IPFS. At last, the auditors commit their decisions to the blockchain via a signed transaction.

Committing the results

The result has the following form:

1. Session ID: Address (the pallet provides this given the address of the interviewee)
2. Result: Number (yes/no/not decidable)
3. Report: List of DDIs
4. Picture: CID

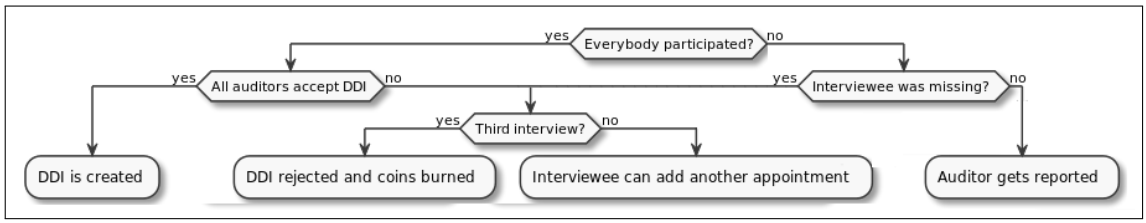

Figure 3.6.: Audit results and outcomes

Different scenarios could occur. Figure 3.6 shows the possible results and their aftermaths. An interviewee who missed an appointment can create another appointment. To avoid a DOS attack, this is possible three times. If an auditor was missing, the auditor should get reported by the other auditors and the interviewee can create another appointment without losing one of the three chances. If an auditor is reported too many times for being absent in interviews, the auditor gets temporarily locked out of interviews. This mechanism impedes the scenario that an auditor does not participate but still transmit results. If the auditors do not accept the DDI in unison, the interviewee can schedule another appointment with different randomly selected auditors, as long as the last interview was not the third chance (excluding session where auditors did not participate in the interview). If after three chances the auditors still not decided for the creation of the DDI in unison, the reserved balance of the interviewee is burned. If the auditors decide in unison, the DDI is created and bound to the picture that was enrolled during the interview. Every auditor who participated and submitted a result is rewarded with new generated coins.

3.3.4. Identity level

The identity level is a value to estimate the likelihood that the identity is real. For an user to obtain identity level x, a DDI verification with x other auditors who are verified level x DDIs must succeed. For example, to obtain identity level one, the interviewees DDI must be validated by one auditor with identity level one. Assuming that 1% are willing to act fraudulent, the chance that a level one identity is not real is 1%. Since the auditor selection process is random, an auditor cannot deliberately validate their own DDI and therefore is not able to spawn an arbitrary amount of false DDIs. To obtain identity level one, the interviewees DDI must be validated by five auditors with identity level five. This means that the auditors themselves previously have undergone an interview with five auditors with identity level five. Assuming that 1% are willing to act fraudulent, the chance that a level five identity is not real is less than $0.01^5 = 0.00000001\%$. Even if 10% are willing to act fraudulent, the chance is still less than $0.1^5 = 0.001\%$. Depending on the impact of the interaction with the blockchain by the DDI, different identity level requirements are appropriate. Besides countering frauds, an increased amount of auditors also increases the chance that fake interviewees are detected.

3.3.5. Potential dangers

Some scenarios could increase the number of invalid identities in the system. One scenario is that the person behind the DDI has aged, had injuries or surgery. To avoid that the person is able to create additional DDIs, a DDI with identity level x is forced to be audited by x other valid DDIs with identity level x periodically, for example every year. The DDI authenticates with a signature whose underlying private key is liked to the address which in turn is linked to the DDI. After the authentication, the auditors find consensus about an optimal enrollment picture and enroll it. They commit their result to the blockchain, which notes which was the old picture CID and which is the new picture CID, such that other auditors and IPFS storage providers can remove the old picture and add the new picture.
Another problem are deep fakes. Those can manipulate a face within a video to look like another face. For the current methods, detectors with near to 100% accuracy exist[Tol+20][Rös+19]. It is highly recommended that such software is used during the first stage of the audit.

3.3.6. Bootstrapping

If no identities exist in the first place, how can an audit be performed to validate new DDIs? The initial DDIs can be included in the first block of the blockchain, the

genesis block. During the test-phases the developers obtain the first DDIs and audit other users. When the blockchain is about to be launched as a main network after the test-phases, another solution must be used to keep up the trustless property of the blockchain. This problem can be solved by initiating a meeting before the blockchain is started. The complete meeting is recorded and every participant is audited by every other participant. Every audit, every resulting DDI and the associated picture and its CID are recorded and published for public access. This way, everybody in the future can verify that the initial DDIs, who are together the ancestor of every other DDI in the system, were honestly audited and created. The members of the meeting can be selected in the following way: Users who have majorly contributed to the previous test of the system are invited to participate. The other half of the invitations are send out to randomly selected users that applied during a public registration phase.

3.4. Council pallet

The council is a small team of elected DDIs with an appropriately high DDI level (5 or greater). Their duty is to review critical decisions and to resolve conflicts.

3.4.1. Election

The council pallet rotates through four major states, which are depicted in Figure 3.7. First the council is elected. When the system is run for the first time, a predetermined council of major contributors serves until the first council is elected. With the genesis block the first election starts, therefore the term of office for the first council is very short. During the first state DDIs with an appropriately high DDI level can submit an application. The application CID is stored on the blockchain and the application itself is stored in the IPFS storage with help of the OffchainWorker pallet.

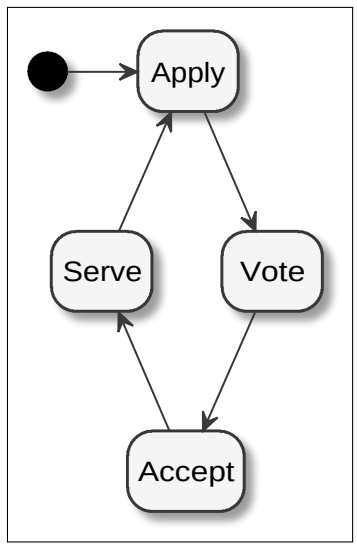

Figure 3.7.: Council states

During the “Vote” state every DDI with an appropriately high DDI level (3 or greater) can vote for one applicant. The circumstance that every human being can only own exactly one DDI and in addition everybody with an appropriate DDI level can participate to vote leads to a truly democratic election. After the vote period has ended, the system selects 1% of the candidates, starting at highest vote ratio and descending into lower vote ratios. The number of candidates selected for the office must be between 16 and 1000. If 1% is less than 16, the difference is used to determine the number of

highest voted candidates that will be selected starting from the candidate with the most votes. If 1% is more than 1000, the difference is used to determine the number of lowest voted candidates that will be deselected from the list of selected candidates, starting at the lowest voted candidate and ascending to higher voted candidates. This mechanism ensures that the council is neither controlled by few DDIs nor overfilled, such that the evaluation time will negatively impact the block production time.
During the next state, "Accept", the selected candidates can either reject or accept the office for the next period. In case a candidate rejects the position, the next candidate with the highest vote ratio on the candidate list is offered the position. When a deadline was reached and the number of selected candidates is within the acceptable range, the system enters the last state before repeating the cycle. In case not enough candidates could be selected, the deadline is postponed. In case there are no candidates left, the system enters the last state. Alternatively the system could move back to the "Apply" state, but this could lead to endless election times with no council and therefore stall the complete system.
During the last state, "Serve", the selected candidates serve the community as the council. Their responsibilities are described in section 3.4.2. The "Serve" state and the other three states overlap. At the end of a councils period of tenure of office, new applications are accepted, voted for and the corresponding candidates can accept their role. Only when the next council is determined, the previous council is suspended.

3.4.2. Responsibilities, rewards and punishment

The council is responsible for resolving conflicts and executing the final vote on sensible decisions. For example, when a project is stalling and was flagged as a fraud, the council must intervene and find out if a fraud is happening, id est the project team or a big fraction of it collectively decided to not produce results but get paid. A sensible decision is for example whether the concerns DDIs had about a proposal are so serious, that the proposal must not be converted to a project and should be revised instead. The pallets from which those conflicts or sensible decisions originate decide how impactful a decision is and how high the vote ratio of the council must be to execute it. Figure 3.8 depicts this situation. A pallet requests a poll from the council pallet and supplies a deadline in form of a future block number x with the request. The council pallet creates a poll with an unique id y and schedules the end of the poll at block number x by using the Scheduler pallet. Every council member can use an off-chain tool, which queries the available polls from the blockchain and separates polls the council member already voted for (by using the DDI) from those which the council member did not vote for. The council member sees when the deadline (including an estimated human readable date and time) for a specific poll is and can vote for every poll that has not

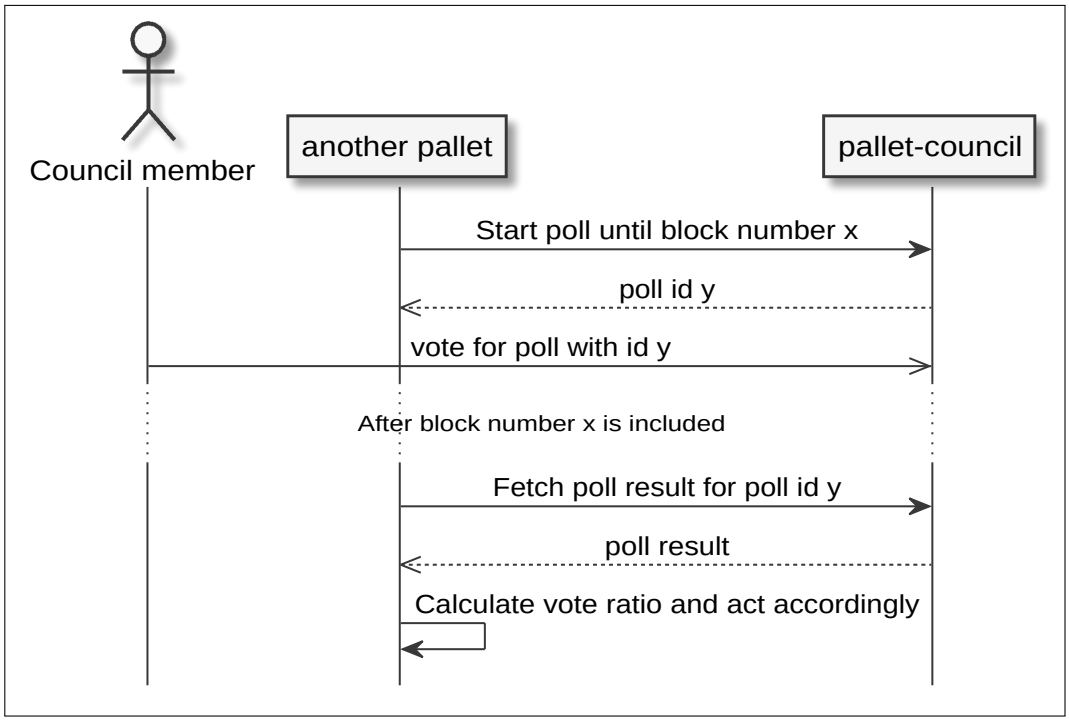

Figure 3.8.: Pallet requests vote from the council and fetches results

reached the deadline yet. The selected option is transmitted to the blockchain as a signed transaction that includes a function call to the voting function of the Council pallet. When the deadline is reached, the poll result can be fetched by anybody given the correct poll id. When the other pallet that requested the poll receives the result by a function call, it can evaluate the result and act appropriately.
Two strategies are combined to motivate the council members to participate. The first strategy uses positive reinforcement. For participation in a poll, the council member is rewarded with new coins. For not participating in a poll, the council member is punished with a strike. After too many strikes, the council member is removed from office for refusal of service. In this case, during the next week the candidates that were not selected for office can signal interest in this role. The candidate who signaled interest and previously received the highest number of votes takes the office.

3.4.3. Emergency election

The emergency election handles a worst-case scenario. Any DDI with an appropriate level (3 and above should be sufficient) can vote to reelect the council at any time. If more than two thirds of the (level 3 and above) DDIs, who interacted with the blockchain by submitting a transaction in the past month, vote to divest the current council of office, the process to determine the next council is started. This includes

traversing the first three states of a council, “Apply”, “Vote” and “Accept”. Only when the next council is determined, the previous council is suspended. This mechanism should ensure that the council does not act in their interest an represent the will of the community. Naturally a community is ready to accept many compromises, nevertheless they should have a last resort in case they are forced to accept too many of them.

3.5. Proposal pallet

The Proposal pallet offers functionality for a community of unique DDIs to express their ideas and concerns about projects that should be funded by the system and implemented by its participants.

3.5.1. States

The pallet rotates through a set of states, which are depicted in figure 3.9. During the first state, identified users can publish proposals. The proposals CID is stored in the blockchain, whereas the original data is stored in IPFS storage. Every DDI can propose one proposal.
In the next state, no proposals can be added anymore, but DDIs can vote from proposal that were carried over into this state. A vote ratio threshold must be specified, that is used to determine which proposals are passed to the next state.
During that phase the community of DDIs can share their concerns about each proposal. Concerns should help to find critical problems early, which might lead to discouraging the conversion to a project and demand for revision.

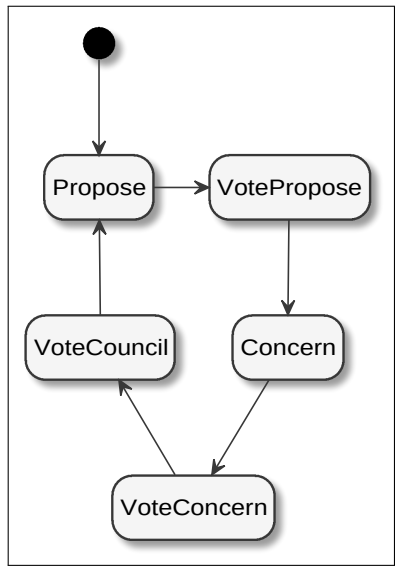

Figure 3.9.: Proposal states

After the concerns were shared, DDIs can vote for them just like for proposals during the second to last state. This mechanism is used to extract a reasonable set of concern with the highest impact. Including every concern is not manageable, because they must be reviewed in the next state and furthermore they consume IPFS storage space.
Last, the community elected council does vote for the conversion of the proposals to a project based on the concerns. Rejections should be reasoned well and should be supported by a supermajority of the council. The collection of votes from the council members is handled by the Council pallet, whereas the evaluation is handled by the Proposal pallet, as described in section 3.4.2. In case a proposal passes all stages, it

is converted into a project by invoking the Project pallet. The DDIs who voted for proposals and concerns that were converted into a project are rewarded with new coins.

3.5.2. Rewards

The DDI who proposed a proposal that got converted to a project is rewarded significantly, because the envisioning of the systems future is an exceptional task, which eventually defines the purpose of the system. Therefore proposing a proposal that gets converted to a project is the only interaction with the system that is rewarded with a percentaged reward, that depends on the total project cost. Votes for proposals are limited to one per DDI. Votes for a proposal that is accepted into the next state are rewarded. DDIs are encouraged to discuss and think about which proposal has the highest positive impact on the system, since every DDI can only vote once per period (one period encompasses the rotation through every state). For every concern that is included into a proposal, the author of the concern is rewarded. In contrast to the reward for accepted proposals, concerns that are included are not rewarded percentaged. DDIs can vote multiple times on either the same or different concerns. For each concern that was included to a proposal, the DDIs who cast a vote for it are rewarded. A proposal usually has more than one concern, leading to the circumstance that more concerns are accepted and consequently the required vote ratio for a concern to be accepted is lower than the required vote ratio for a proposal. Those two actualities lead to the decisions, that votes for concerns that are included in proposals should be rewarded less than votes for accepted proposals. As described in section 3.4.2, council members are rewarded for casting their votes in a poll.

3.6. Project pallet

The project pallet manages projects that were spawned from community elected proposals. It offers functionality to arrange a team, run the project and at last to finalize it. This section is only a guideline and not complete. Many corner cases are not described here and most likely will lead to an insecure system.

3.6.1. Roles

Figure 3.10 shows the three different roles in a project. The first participant in a project is the project leader, whose tasks are to evaluate the required roles, assemble a team and manage the project development path, such that goals are reached depending on their importance. Of course, any activity a project leader in a usual digital production environment does practice is also included. A worker helps realizing the project by utilizing skills that were denoted in the workers application and requested in the job

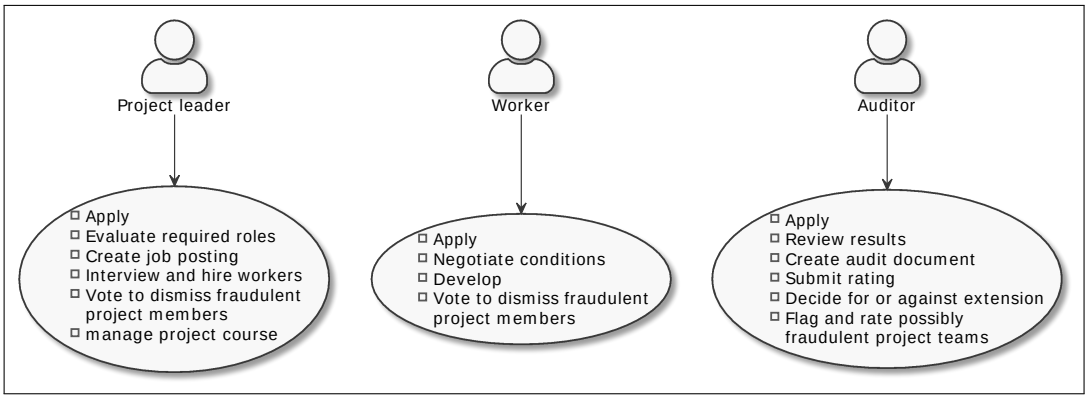

Figure 3.10.: Roles in a Project

posting. Both the project leader and the worker must report fraudulent behavior to the system. The last role is the auditor role, who inspects the quality of the final results, creates an audit document, submits a rating, flags and rates flagged and possibly fraudulent project teams and helps to decide whether a project should be prolonged, in case it was not completed in time.

3.6.2. States

After another pallet has spawned a new project, in this case the Proposal pallet, a state machine for that specific project is started. The state machine is depicted in figure 3.11. During the first state a project is empty. Nothing but the proposal and the associated concerns exist. For a fixed period of time, project leaders (who are ultimately DDIs with an high identity level) can publicly apply for the position.
In the second state, DDIs can vote for a project leader based on the existing applications. The application is a collection of documents that consists of the same documents a classical application consists of. In addition, the application includes the estimated cost for the project. The Project pallet offers a scoring functionality, which helps the voters to decide for a project leader. For example, a complex and crucial project might have many applications. In such a case a project leader with more successfully completed projects on the platform and therefore a higher (trust-) score might be preferred compared to a project leader with no proven results on the platform, even though the project leader with the higher score is more expensive. In contrast to a

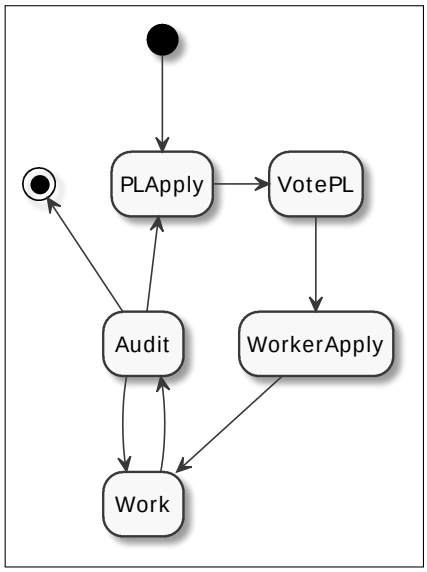

Figure 3.11.: States of a specific project

crucial and complex project, a simple project is a better fit for first-time project leaders who demand less capital to prove themselves. The winning project leader gets the responsibility for the project, which from that moment on is supplied by a pool of newly generated coins. Any user that voted for the project leader does get rewarded.
During the next state, the project leader publishes job offers, describing the required skills for the role. Workers can apply for those roles. A worker is a DDI with a specific skill set who can contribute to a project. The worker and the project leader reach an agreement about the monthly payment, which is immutably stored in the blockchain. The project pallet allocates the coins as stipulated by contract between project leader and worker, consequently the project leader has no control over those funds anymore. The remaining coins are allocated for the project leader and partially paid out in monthly intervals. After every role is filled, the project begins. The pallet handles payouts as stipulated by contract for every participant.
With the second to last state the working period begins and all participants are on equal footing. If the participants in the project decide that a colleague is restraining the progress in any way, they have the option to vote off the colleague by a supermajority vote with 66% or more votes. The original job offering created by the project leader is used in case it is a worker role, with the exception that the remaining coins are a fixed payout for the remaining project duration and are not negotiable anymore. The project leader must approve or reject the worker. This is the only situation where the project leader does have more privileges than the workers. If a project leader is voted off, other project leaders can apply for a short period of time. The workers can vote for a new project leader, although the options are limited to the project leaders with the highest scores.
During the last state, a final inspection happens. This part is poorly evaluated in this work and requires further examination. Nevertheless, a suggestion is presented in section 3.6.3

3.6.3. Final audit

For the inspection of a project a project auditor working group is required. To become an auditor, the DDI must have passed a certain score threshold as a project leader or worker. Initially, some of the core developers are elected to fill those positions. For every audit, multiple auditors must provide an audit document with a short summary. The auditors can register themselves for a project, the selection depends on the score of the auditor. A system is required that matches the projects importance to the score of the auditor and therefore is able to give auditors with lesser scores a chance to improve their score. Other auditors can either like or dislike the auditors summary and conclusion in exchange for a small but appropriate reward. This is used to determine

the score gain of the auditors who review the project. In principle, the proposed idea is a scoring system based on peer-reviews. This is by no means a solution and should only serve as suggestion for further exploration. Depending on the outcome of the audit, the participants of the project are rated adequately. It is in their interest to deliver a good result, otherwise they reduce their chances to participate in future projects (since every person can only own one DDI). In case a project was not finished in time, the auditors also supply their assessment whether or not the team is capable to satisfactorily complete the project. In case of rejection by a majority of the auditors, the project team is dismissed and the state is reset to the first state during which project leaders can apply.

3.6.4. Fraud and intervention

Every auditor of the auditor working group can flag a project as fraudulent with a short summary. After a certain threshold is reached, the top high scoring auditors are asked to share their opinion. After a deadline was reached, for example one week, the ratio of high scoring auditors who approved the flag to high scoring auditor who declined the flag is evaluated. If a certain threshold is reached, the case including the opinions of the auditors is passed to the council. The council is the last instance to approve the disbandment of the project group by reviewing the opinions of the auditors and voting for or against the disbandment. The council represents a neutral group, that is integrated in the process to circumvent potential biased disbandments. The score of every auditor who submits a flag is either increased or decreased, depending on whether the flag leads to a disbandment. This should motivate auditors to carefully use this feature. Every participant in this process is rewarded with an appropriate amount of coins for correct decisions, including the auditors who flagged the project.

3.6.5. Project lifetime

The source code is open source and the development process is completely transparent. After the project was successfully finished, it becomes a public good. The project stays open source and is available to everyone. In case the operation of the resulting product requires a lot of computation power or storage space, the OffchainWorker pallet can be used to register storage and computation time requirements for this project. IPFS hosts and computation time providers can register themselves for operation of this project in the OffchainWorker pallet. The pallet ensures, that they are rewarded adequately for computation time and storage space they provide to operate projects that were created in this system. This creates a natural balance between supply and demand. If there is more demand than supply, additional computation time providers or storage providers see this opportunity to effectively offer their service in return for a reward and register

in the OffchainWorker pallet. If there is a lower demand than supply, providers might decide to switch to a more profitable project. Irrespective of this fact, some enthusiasts might still operate the project as providers, such that a project only dies when there is no more belief in the project.

3.6.6. Runtime upgrade projects

The Substrate framework allows to change the state-transition function via forkless runtime upgrades. A proposal might include some changes to the core of the system that require such an upgrade. In this case, only the highest scoring project leader, workers and auditors may participate. The council must approve the changes by voting for or against the application of the changes. The agree to disagree ratio must be extraordinary high, because the application of a runtime upgrade changes the system at its core.

3.7. OffchainWorker

The OffchainWorker pallet provides functionality to register storage and computation requirements and rewards providers for supplying storage capacity and computation time.

3.7.1. Essential problem

The major problem in designing this pallet is that the verification of the services, id est whether the storage providers and computation time providers really provide those services, cannot be part of the state-transition function of the system. This is due the circumstance that the verification time is not deterministic and therefore not compatible with a blockchain that targets a constant block time. For example, to verify whether a storage provider really provides the data the storage provider claims to provide, the system is forced to download the complete storage from the storage provider, calculate the hash of the received data and compare it to the hashes which are stored in the local storage trie. This is not compatible with the requirement for a constant block time. Additionally, consensus rules require that the state transit must be verifiable by any other node and to reach consensus, the outcome must be equal for every node, therefore every node in the network must download the complete storage of the storage provider and verify it. Even if the previous two problems would be solvable, still the requirement stands that at any time anybody should be able to verify whether a block is valid or not. When a block is verified that queried a service provider in the past, it might be that the service provider does not provide the service anymore. Therefore future verification of the block would lead to different results than in the past and the node would discard the

block as invalid, which would lead to a fork. This is obviously unpractical. The same applies to computation time providers. Every node of the system would be required to execute every service every "Computation time Provider" (CTP) offers and verify the results. Even worse in this case, the request must differ every time, otherwise the CTP can simply return the precalculated result and in reality don't really offer this service. Consequently the verification of storage and CTPs must happen outside of the blockchain and only the result should be provided to and optimally verified by the blockchain. This comes at a major cost: The system is not trustless in this aspect, because it is not verifiable whether or not the storage provider and CTPs really provide the services they claim to provide. New ideas and methodologies are required.

3.7.2. Computation

Figure 3.12 shows a way of potentially realizing an offchain CTP. Once an application is registered in the OffchainWorker pallet, DDIs can register as CTPs for the application. This includes a (preferably static) ip address (and in case of ipv4 a port). The user who requests computation time for that application from a provider supplies a timestamp and a signature. The CTP fetches a list of DDIs from the Identity pallet and verifies that the public key behind the signature does belong to a known CTP. This is relevant for the CTP, because as explained a little later in this section, the system will reward the CTP if enough unique DDIs used the service. If the signature belongs to a valid DDI, the CTP computes the result and returns it to the user, otherwise the request is rejected. The user should verify the result and return a signature over the provider DDI, the application name and the current timestamp. This is not enforceable. The user already got the result and may just decide not to return a signature over this data. In that case the CTP can ban the DDI and not process any request from this particular DDI in the future. Once per day, every CTP can transmit a set of signatures over the DDI, the application name and the timestamp the request was fulfilled. This list should be capped to a predefined amount of entries. The system verifies that the timestamps are not from the past, matches whether the CTP DDI is listed for the application that was specified and whether the signature is from a different known DDI. If a predefined amount of unique DDIs signed such request, a predefined reward per day is rewarded to the provider.
This approach has some drawbacks. For example, the CTP could pay other DDIs for just signing the text without computing the result and promise them a share for the reward. Luckily, if the required DDI level to use CTP services is high enough (2 should be sufficient), it is too unlikely that the CTP could obtain multiple DDIs and create enough valid signatures. To keep the the number of CTPs who fraudulently pay other DDIs in exchange for signatures down, a mechanism is required that could lead to a

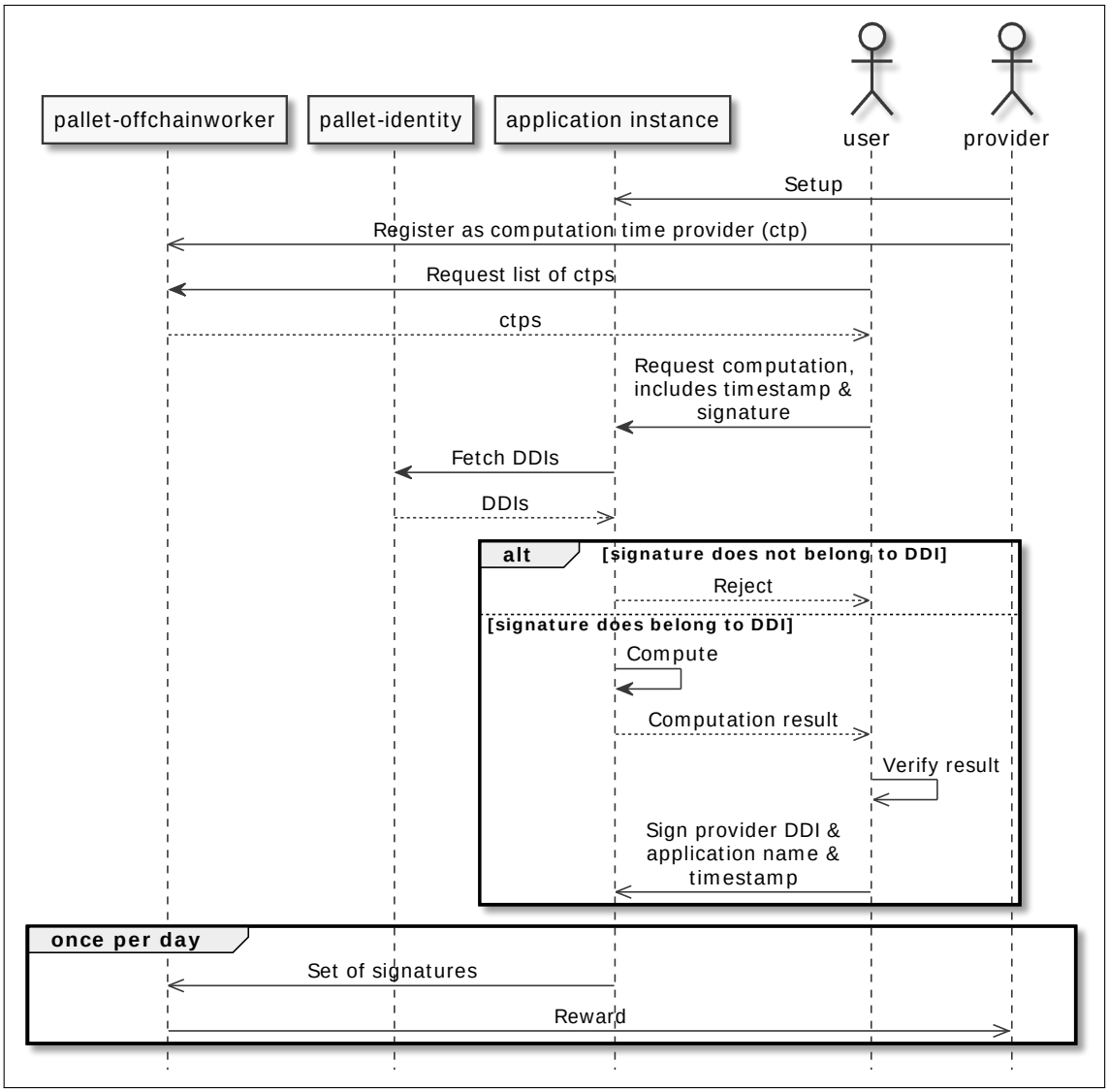

Figure 3.12.: Computation time provider: Registration, delivery of service and reward

lock-out. Such a mechanism could be for example, that higher level DDIs voluntarily (without a reward) can flag a CTP for not providing the services the CTP is rewarded for. When many DDIs flagged the CTP, for example 1% of the DDIs that have the required level or higher, the DDI behind the CTP receives a permanent lock-out from operating as a CTP. Another mechanism can be that DDIs with an appropriate level can vote multiple times without charge whether or not CTPs offer the service they registered for (1 vote per CTP allowed). Only CTPs with a vote-ratio greater than 0.5 will get paid out.

3.7.3. Closing note

The problematic to check providers and reward them turned out to be a very difficult problem. Due to the fact that the investigation can not be done on-chain and therefore is not part of the state-transition function, major problems occur. To give an idea how a future solution could possibly handle CTPs and storage providers: The system could rely on unique DDIs with an appropriate level to vote certain times per day for free in the OffchainWorker pallet. Users can either approve or decline that a service provider really provides the service. Service providers with a negative vote ratio would not get paid out, whereas service providers with a positive vote ratio would get paid out. This still raises the question, how to fairly pay the providers. For example, some storage providers might supply 10TB of storage space, some only 100GB. Some storage providers might offer a total bandwidth of 10gbit/s, whereas some might only offer 10mbit/s. Some computation time providers might offer 1TFLOPS, whereas some might offer 80TFLOPS. The question remains open, how to differentiate their contribution and reward them appropriately. The whole off-chain service provider problematic would be well suited as a topic for a complete master thesis.

4. Implementation

This chapter describes the PoC implementation of the system described in the design chapter. Due to the complexity, only the Proposal pallet is fully implemented, every other pallet only offers an interface, type definitions, structures and a dummy implementation of that interface. Even with that constraint, the number of custom lines of code written during this thesis amounts to 1,959 (1,413 without comments). This is due to the nature of the logic that is part of the state-transition function. Many corner cases must be regarded, as will be shown later in this chapter, otherwise DOS attacks can be executed to halt the blockchain or a security issue could lead to the spurious issuance of coins.
The chapter starts by presenting an overview of the project structure in section 4.1, followed by the description of the Identity pallet (section 4.4), the Council pallet (section 4.5), the Project pallet (section 4.6) and the Proposal pallet (section 4.7).

4.1. Project structure

The project is located in the following path on the CD or digital archive supplied with this work: implementation/superorganism. The project is a template of the Substrate project[1] whose version is fixed at v2.0.0[2]. Substrate is written almost exclusively in Rust, a performant low-level language that allows the developer to achieve memory-safety and thread-safety at compile-time. For this project, rustc 1.46.0 (04488afe3 2020-08-24) and rustc 1.49.0-nightly (a1dfd2490 2020-10-05) were used. To be able to created a WASM blob that can be used to execute forkless runtime upgrades, the target "wasm32-unknown-unknown" must be added to rust-nightly. Substrate is modular, which suits well to the inherent package management system offered by Rust. The code that is assignable to a distinct feature is separated and included in its own library, which in turn is included in a package. Such packages can contain a distinct hashing or signature algorithm, a distinct encoding scheme or a pallet for example. The template project is delivered with a "Cargo.lock" file, which enforces a fixed version number for

[1] `https://github.com/paritytech/substrate`
[2] `https://github.com/substrate-developer-hub/substrate-node-template/tree/655bfdc6fc922d117849cbcf808ee5bf2dfa1d53`

each package that is used. Any packages that are included in development from that point on, are mentioned in the "Cargo.toml" file.

Figure 4.1 depicts an overview of the project structure. The project consists of two

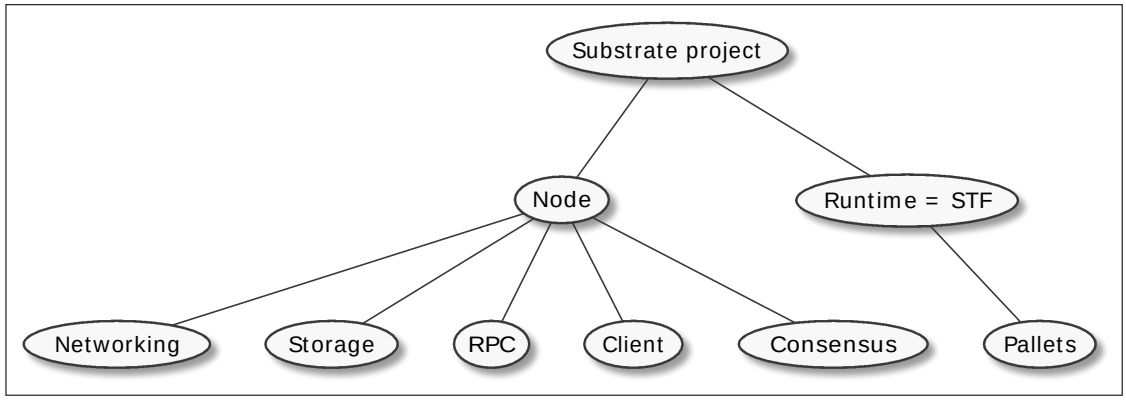

Figure 4.1.: Overview of the project structure

essential sub-projects, the "node" project and the "runtime" project. The node project contains the core functionality of the Substrate node, which includes but is not limited to the low-level networking, management of storage (merkle-patricia trie abstraction into a database), client and "Remote Procedure Call" (RPC) functionality and consensus, that dictates whether or not a transaction and a block are valid and which block is selected to extend the blockchain and consequently, what the next parameters for the "State Transition Function" (STF) will be.

The runtime project does manage the business logic of the blockchain. Therefore runtime is a STF, id est it dictates how the state of the blockchain (the merkle-patricia storage trie) is changed in respect to a block and inherently every transaction within it. The runtime is mainly a collection of pallets, which extend the STF. The runtime also defines global default types and constants used by the pallets.

4.2. Runtime

The runtime is specified in the file that is located at `implementation/superorganism/runtime/lib.rs`. There is a difference compared to the composition of pallets described in the design chapter. Figure 4.2 depicts which pallets were removed (transparent) and which pallets were added (white arrow) instead. The PoS pallets have been replaced with a PoA "Proof-of-Authority" pallet. The configuration and operation of a PoA system is not that time-consuming and cumbersome compared to a (N)PoS system. It is completely sufficient that a PoC implementation uses a fixed authority set to produce blocks. A system that is publicly accessible must not use PoA as the block authoring mechanism, otherwise the system is not permissionless and not trustless. In addition,

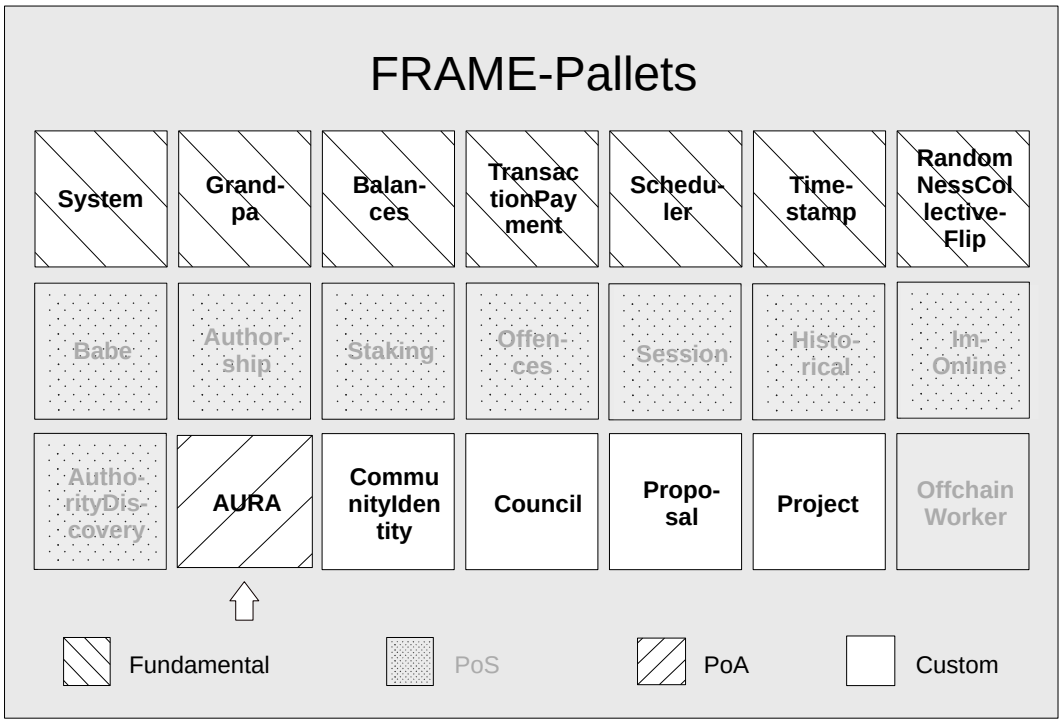

Figure 4.2.: FRAME-pallets: Differences from the pallets specified in the design chapter

the OffchainWorker pallet is also not part of the PoC implementation. Throughout the implementation it is assumed, that the data exists in the IPFS storage. Consequently, the pallets only store the CIDs without invoking the OffchainWorker pallet to signal the occurrence of new data.
The "Authority-Round" (AURA) consensus mechanism is one of the simplest consensus mechanisms. Every member in a predefined set of authorities is selected by rotating through the set using the round-robin selection rule. During one slot, the selected authority can author the next block. The block is appended to the canonical chain (in this case the longest valid chain). Every slot is valid for t seconds, which approximately matches the desired block production time.

4.2.1. Structure

The source file can be separated into five sections:

1. Inclusion of dependencies
2. Definition of global types and constants
3. Definition of pallets: Constants and integration of modules that implement required traits

4. Inclusion of the pallets into the runtime and specification of their interaction with it

5. Implementation of runtime APIs.

In the first section any dependency is included, for example the custom pallets that were created as part of this work. Listing 4.1 shows some "lines of code" (LOC) from that section.

```
pub use pallet_community_identity;
pub use pallet_council;
pub use pallet_project;
pub use pallet_proposal;
```

Listing 4.1: Runtime: Section 1 - dependencies

In the next section types and constants are defined, for example the block number type, accountid (address) type or the estimated block time, slot duration for AURA slots and minutes expressed in blocks, as shown in listing 4.2.

```
pub type BlockNumber = u32;
pub type AccountId = <<Signature as Verify>::Signer as IdentifyAccount>::AccountId;
pub const MILLISECS_PER_BLOCK: u64 = 6000;
pub const SLOT_DURATION: u64 = MILLISECS_PER_BLOCK;
// Time is measured by number of blocks.
pub const MINUTES: BlockNumber = 60_000 / (MILLISECS_PER_BLOCK as BlockNumber);
```

Listing 4.2: Runtime: Section 2 - types and constants

After that every pallet is configured by either defining constant values with an appropriate type using the `parameters_types!` macro or by assigning an implementation of a trait. Listing 4.3 shows how the proposal pallet that was created during this work does bind some other pallets that implement a certain trait, including the Identity, Council and Project pallets that were created during this work. It also shows how a constant is defined and bound to variable. In case of constants, the required trait does demand that a getter function is implemented. `parameters_types!` does derive the getter function.

```
parameter_types! {
	pub const ProposeCap: u32 = 1_000;
	//	...
}

impl pallet_proposal::Trait for Runtime {
	type Currency = pallet_balances::Module<Runtime>;
	type Scheduler = pallet_scheduler::Module<Runtime>;
	type Identity = pallet_community_identity::Module<Runtime>;
	type Council = pallet_council::Module<Runtime>;
	type Project = pallet_project::Module<Runtime>;
	type ProposeCap = ProposeCap;
	//	...
}
```

Listing 4.3: Runtime: Section 3 - constants and dependencies

```
construct_runtime!(
	pub enum Runtime where
		Block = Block,
		NodeBlock = opaque::Block,
		UncheckedExtrinsic = UncheckedExtrinsic
	{
		//	...
		Proposal: pallet_proposal::{Module, Call, Storage, Event<T>, Config},
}
```

Listing 4.4: Runtime: Section 4 - Inclusion of the pallets into the runtime

In the next section the runtime is constructed and the interaction of the pallet and the runtime is specified. Listing 4.4 shows how the Proposal pallet is included into the runtime. By including the `Module` struct from the proposal pallet in the runtime definition of Pallet, every (public) function is imported and therefore available for every component in the runtime. By specifying the `Call` enum, the runtime knows which functions can be called by a transaction. Since the `Storage` struct is imported, the runtime knows that and how the pallet uses the storage. By importing the `Event<T>` type, the runtime knows that events are used and which custom types are used within those events. Because the `Config` struct is imported, the runtime knows that during the creation of the genesis block, the modules storage must be altered according to the rules specified in the pallet.

The last section implements runtime APIs that were previously declared within the `sp_api::decl_runtime_apis!` macro. Listing 4.5 shows a snippet, that uses the

`random_seed` function provided by the RandomnessCollectiveFlip pallet to provide a seed to the `sp_block_builder` package, which uses it as a seed for random number generation during block creation.

```
impl_runtime_apis! {
        impl sp_block_builder::BlockBuilder<Block> for Runtime {
                fn random_seed() -> <Block as BlockT>::Hash {
                        RandomnessCollectiveFlip::random_seed()
                }
}
```

Listing 4.5: Runtime: Section 5 - Implementation of previously declared runtime APIs

4.3. Structure of a pallet

In this section the structure of a pallet is described. Substrate provides many macros which ease development by generating boilerplate code. While the original Proposal pallet source code without comments (and structure, enum and type definitons) has 583 LOC, the expanded code generated by the macros has 3788 LOC. Substrate offers macros to define the storage structure, including the definition of how the storage is altered during genesis. It also offers macros to define events, errors and the module definition that results in functions that are callable via extrinsics (mostly signed transactions). The following sections show examples of those macros and explain what they do by using the proposal pallet as an example.

4.3.1. Type declarations and loosely and tightly coupled pallets

```
pub trait Trait: frame_system::Trait {
        type Currency: ReservableCurrency<Self::AccountId>;
        type ProposeCap: Get<u32>;
        //  ...
}
```

Listing 4.6: Pallets: Type declaration and loosely and tightly coupled pallets

Listing 4.6 shows a snippet of the Trait definition of the Proposal pallet. In the first line, tightly coupled pallets can be specified, just like it is done with the frame_system pallet. By requiring that the pallets Trait trait (the uppercase variant is the trait name, and the lowercase variant is the interface concept name in rust) does implement the frame_systems Trait trait, the Proposal pallets Trait trait is bound to the

specific implementation of the frame_systems Trait trait. In other words, the pallet does require the specific implementation the frame_system offers, any other implementation is not allowed. This is an inflexible but a definite way to incorporate another pallet. Everything that follows in the scope of the trait definition is exactly what is bound to actual values within the runtime pallet, as shown in listing 4.3. In contrast to the tightly coupled frame_system pallet, the `Currency` type only specifies a Trait, `ReservableCurrency<Self::AccountId>`, that must be implemented by another module. Any rust module that implements this trait can be bound to the type in the runtimes pallet specification, just like in line 7 of listing 4.3. To define constant values, the `Get<T>` trait is used, where `T` is the type to get. Those constants are defined in the runtimes pallet specification, like in line 2 of listing 4.3 and assigned to the type like in line 12 of listing 4.3.

4.3.2. Storage

Listing 4.7 shows a snippet of the storage declaration in the Proposal pallet. The first keyword specifies the visibility. A public (`pub`) storage can be accessed and modified by other pallets in the runtime. After that the name of the structure whose functions are accessed to modify the storage element is defined. The structure is completely generated by the macro. Followed by that, three possible options can be specified: `get`, `config` and `build`. `get` is used to generate a getter function, `config` and `build` are used to initialize the storage during genesis. Next the type of the storage element is defined. If empty, one value can be stored in the storage element. In addition, it is possible to set the storage element type to an hashmap and specify that hash function to be used. Using this option it is possible to store a key and value tuple with predetermined types, as seen in the example. Given a key, the value is returned. The system also offers the option to set the double hashmap, in that case also the value can be supplied to retrieve the corresponding key. At last, a default value can be applied to the storage. Those three storage types and the chance to initialize the storage offer enough functionality to realize any storage requirements. Since one-to-one mappings are the only option to store associated items, the design of a working and especially efficient storage layout can be cumbersome.

```
decl_storage! {
	trait Store for Module<T: Trait> as Proposal {
		pub State get(fn state) config(): States = States::Uninitialized;
		pub Proposals get(fn proposals): map hasher(identity)
			IdentityId<T> => Vec<Proposal> = Vec::new();
		//	...
	}
}
```

Listing 4.7: Pallets: Storage declaration

4.3.3. Events

The `decl_event!` macro allows the pallet author to add custom events to the pallet. An event is a special storage entry in the pallets event storage, that is composed of a unique name (per pallet) and optionally a n-tuple. Events are used to inform users and applications that an event occured in a pallet. The developers can freely specify which events exist and when they are emmited. A part of the event declaration of the proposal pallet is shown in listing 4.8. The example begins with the `decl_event!` macro, followed by the declaration of the event enum including custom types used in the enum. In this case, `BalanceOf<T>` is specified for example. `BalanceOf<T>` is a type that was previously defined as the balance type the associated pallet that supports the proposal pallet with an implementation of the `ReservableCurrency` trait uses for balances. Within the body of the enum, an event with the name `TotalProposalReward` that contains a singleton (1-tuple) of the previously defined `Balance` type is used. Whenever the pallet emits this event, listeners know that a total proposal reward of `x Balance` was issued.

```
decl_event! {
	pub enum Event<T> where Balance = BalanceOf<T>,
	                        ID = IdentityId<T>,
	                        PW = ProposalWinner<IdentityId<T>> {
		///	\[Balance\]
		TotalProposalReward(Balance),
	}
}
```

Listing 4.8: Pallets: Event declaration

4.3.4. Errors

Custom errors for dispatchable calls, id est calls that came with an extrinsic (most of the time a signed transaction), are defined by using the `decl_error!` macro. Listing 4.9 contains an example. Whenever a previously known error occurs during the execution of the pallet, the pallet can return one of the default errors provided by the framework or one of the errors defined in the error enum within the `decl_error!` macro. The error provided as an example in listing 4.9 would be returned, when a transaction is submitted and executed that tries to propose a proposal that was already proposed before.

```
decl_error! {
        pub  enum Error for Module<T: Trait> {
                ProposalAlreadySubmitted,
                //    ...
        }
}
```

Listing 4.9: Pallets: Error declaration

4.3.5. Module definition

The complete logic of the pallet is defined in the implementation of the `Module` struct that uses the trait Trait, that was declared previously (see section 4.3.1), as its generic parameter. Direct implementation of the `Module` struct enables the developer to add functions, that can be invoked either within the pallet exclusively or in case the function is public, also by other modules that are part of the runtime. In addition, the `decl_module!` macro is used to specify dispatchable functions, that can be called by any module within the runtime and by extrinsics. Listing 4.10 shows an example of a module declaration. Although the macro is called "declare module", the complete logic is included and therefore it is rather a module definition. The first line after the `decl_module!` statement means that any function implemented in this module should be included in the Call enum for this module. As described in section 4.2.1, this means that the functions within become available to extrinsics. As shown in the example, any constants that were defined in the `parameter_types!` macro and assigned in the implementation of the Trait of a specific pallet for the runtime (see listing 4.3) are assigned to a constant that can afterwards be used within the implementation of the module. After the constants the dispatchable functions are implemented. As shown in listing 4.10, the functions that are callable by an extrinsic (in this case the function `state_transit`) return a DispatchResult. This is an enum that can contain a value

to either signal that the execution was successful or erroneous. The errors that can be returned include the errors specified by the `decl_error!` macro. The annotation above the function `#[weight = ...]` defines the weight consumption by this block, which ultimately is used to calculate how long it takes to execute the function and therefore if it should be included in a block or if its inclusion does lead to exceeding the desired block time. Any `ensure_...` command is used to check preconditions that must be met before the function should make changes to the global state. In this case, the function checks whether call originates from another module within the runtime.

```
decl_module! {
        pub struct Module<T: Trait> for enum Call where origin: T::Origin {
                const ProposeCap: u32 = T::ProposeCap::get() as u32;

                #[weight = 10_000 + T::DbWeight::get().reads_writes(5000,3)]
                fn state_transit(origin) -> DispatchResult {
                        // check and change the current state
                        ensure_root(origin)?;
                        Self::do_state_transit()
                }
        }
}
```

Listing 4.10: Pallets: Module definition

4.4. Identity

This section describes the implementation of the Identity pallet as described in section 3.3. The implementation offers an appropriate generic trait definition, necessary types and a dummy implementation of the provided trait. The dummy implementation assigns a valid DDI id with DDI level 5 to every user. The source files are located at `implementation/superorganism/pallets/community_identity`

4.4.1. Trait

Listing 4.11 contains the trait definition of the Identity pallet. The trait has one generic type and four associated types. The generic type `ProofData` defines the type of the data that is stored as a proof. In dependence of the generic type, five additional types are set: The address type, the ticket type, the timestamp type, the identity level type and the identity id type. For demonstration purposes, the system uses a fixed-length array to represent 32 bytes of data as the proof type: `[u8; 32]`.

```
pub trait PeerReviewedPhysicalIdentity<ProofData>
	where ProofData: Codec + Clone + Debug + Decode + Encode + Eq + PartialEq
{
	type Address: Codec + Clone + Eq + EncodeLike + Debug;
	type Ticket: Codec + Clone + Eq + EncodeLike + Debug;
	type Timestamp: AtLeast32Bit + Parameter + Default + Debug + Copy;
	type IdentityLevel: Num;
	type IdentityId: Codec + Clone + Eq + EncodeLike + Debug;

	/// Request a peer review to gain a specific IdentityLevel
	fn request_peer_review(user: Self::Address, identity_level: Self::IdentityLevel,
		at: Self::Timestamp) -> Result<Self::Ticket, DispatchError>;
	/// As a reviewer, approve a reviewed PhysicalIdentity by supplying a proof
	fn approve_identity(review_process: Self::Ticket, proof_data: ProofData)
		-> Result<(), DispatchError>;
	/// As a reviewer, reject a reviewed PhysicalIdentity
	fn reject_identity(review_process: Self::Ticket) -> Result<(), DispatchError>;
	/// As a participant, report a missing participant
	fn report_missing(review_process: Self::Ticket, missing: Vec<Self::IdentityId>)
		-> Result<(), DispatchError>;
	/// Get the appointments for a DDI (when the DDI has to participate in an audit)
	fn get_appointments(identity: &Self::IdentityId)
		-> Vec<(Self::Timestamp, Vec<Self::IdentityId>)>;
	/// Receive the identity level of a specific PhysicalIdentity.
	fn get_identity_level(identity: &Self::IdentityId) -> Self::IdentityLevel;
	/// Get IdentityId for an address
	fn get_identity_id(address: &Self::Address) -> Self::IdentityId;
	/// Get (main) address for an IdentityId
	fn get_address(identity: &Self::IdentityId) -> Self::Address;
}
```

Listing 4.11: Identity: Trait definition

The purpose of the functions is well documented by comments and therefore is not described further, please refer to the referenced listing for information in that aspect. In contrast to the design of the Identity pallet in section 3.3, the result the auditors commit after an audit is split and assigned to three functions: One to approve an identity, one to reject an identity and one to report a missing participant. The implementation still demands for an off-chain application, that connects the users in a decentralized fashion and offers functionality to find participants based on their DDI id. In addition, a peer-to-peer video conference tool with the required functionality as described in section 3.3 is still missing.

4.4.2. Types

The Identity pallet offers two default types, one for proofs (listing 4.12) and one for internal storage of the DDI id, identity level and associated proof (listing 4.13). Those types are not used yet in the dummy implementation. They were created to serve as an example and template for future implementation of this pallet.

```
pub struct PhysicalProof<Timestamp, ProofData> where
        ProofData:   Codec + Clone + Debug + Eq + PartialEq,
        Timestamp:  AtLeast32Bit + Parameter + Default + Debug + Copy,
{
        proof:    ProofData,
        date:    Timestamp,
}
```

Listing 4.12: Identity: Proof type template

```
pub struct PhysicalIdentityData<Timestamp, AccountId, ProofData> where
        ProofData:   Codec + Clone + Debug + Eq + PartialEq,
        AccountId:   Codec + Clone + Debug + EncodeLike + Eq,
        Timestamp:  AtLeast32Bit + Parameter + Default + Debug + Copy,
{
        identity:     AccountId,
        level:     IdentityLevel,
        proof:     PhysicalProof<Timestamp, ProofData>,
}
```

Listing 4.13: Identity: DDI storage type template

4.5. Council

This section describes the implementation of the Council pallet as described in section 3.4. The implementation offers an appropriate generic trait definition and a dummy implementation of the provided trait. The dummy implementation allows another pallet to add a poll, but the function used to fetch the result does always return the same result (6/6 council members vote yes). The source files are located at `implementation/superorganism/pallets/council`

4.5.1. Trait

Listing 4.14 contains the trait definition of the Council pallet.

```
pub trait Council
{
	type IdentityId: Codec + Clone + Eq + EncodeLike + Debug;
	type Ticket: Num;
	type BlockNumber: Codec + Clone + Debug + Eq + PartialEq;
	type DocumentCID: Codec + Clone + Debug + Eq + PartialEq;

	/// As an identified user, vote for a council member
	fn vote_council_member(voter: Self::IdentityId, candidate: Self::IdentityId)
		-> Result<(), DispatchError>;
	/// As an identified user, vote to reelect council
	fn vote_reelect_council(voter: Self::IdentityId) -> Result<(), DispatchError>;
	/// As an identified user, vote to reelect a specific council member
	fn vote_reelect(voter: Self::IdentityId, member: Self::IdentityId)
		-> Result<(), DispatchError>;
	/// As root, queue a poll
	fn add_poll(documents: Vec<Self::DocumentCID>, until: Self::BlockNumber)
		-> Result<Self::Ticket, DispatchError>;
	/// As a council member, vote for a poll
	fn vote_poll(member: Self::IdentityId, poll: Self::Ticket, accept: bool)
		-> Result<(), DispatchError>;
	/// Retrieve result of a poll
	fn get_result(poll: &Self::Ticket) -> Option<Vec<(Self::IdentityId, bool)>>;
}
```

Listing 4.14: Council: Trait definition

The trait demands from an implementation to offer functionality to cast a vote for a nominee and vote to reelect the complete council or a single member of the council. In addition, it demands that the implementation should be able to add a poll, offer a function to vote for the poll as council member and to get the result. In practice the `IdentityId` type allows any type that has the same properties as the intrinsic address type, including the address type itself. Nevertheless, the design of this system instructs to use a proper DDI management pallet, consequently the Identity pallet and its identity type are used.

4.6. Project

This section describes the implementation of the Project pallet as described in section 3.6, with the exception that the trait does not offer an interface for audits. The implementation offers an appropriate generic trait definition, a dummy implementation of the provided trait and custom types. The dummy implementation allows to

spawn a project from a proposal and its associated concerns. The new project is stored in the storage of the pallet and is retrievable at any time. The logic to apply as a project leader or worker, vote for the replacement of the complete team or one single member and the final audit, is still to be implemented. The source files are located at `implementation/superorganism/pallets/project`

4.6.1. Trait

Listing 4.15 and listing 4.16 contain the trait definition of the Project pallet.

```
pub trait ProjectTrait
{
	type Balance: Codec + Clone + Debug + Eq + PartialEq;
	type IdentityId: Codec + Clone + Eq + EncodeLike + Debug;
	type ProposalWinner: Codec + Clone + Eq + Debug + PartialEq;
	type Project: Codec + Clone + Debug + Eq + PartialEq;

	/// As root, spawn a project from a proposal
	fn spawn_project(proposal: Self::ProposalWinner)
		-> Result<Self::Project, DispatchError>;
	/// As an identified user, apply as project leader
	fn application_project_leader(who: Self::IdentityId, project: ProjectID,
		application: DocumentCID) -> Result<(), DispatchError>;
	/// As an identified user, Vote for project leader
	fn vote_project_leader(voter: Self::IdentityId, pl: Self::IdentityId,
		project: ProjectID) -> Result<(), DispatchError>;
	/// As a project leader, open positions
	fn open_position(pl: Self::IdentityId, project: ProjectID, position: DocumentCID)
		-> Result<(), DispatchError>;
	/// As an identified user, apply for a position
	fn apply(applicant: Self::IdentityId, project: ProjectID, position: DocumentCID,
		application: DocumentCID) -> Result<(), DispatchError>;
	// As a project leader, accept application and offer salary
	fn offer_applicant(pl: Self::IdentityId, applicant: Self::IdentityId,
		project: ProjectID, position: DocumentCID, application: DocumentCID,
		salary: Self::Balance) -> Result<(), DispatchError>;
	// As an applicant, accept an offer
	fn accept_offer(applicant: Self::IdentityId, project: ProjectID,
		position: DocumentCID, salary: Self::Balance) -> Result<(), DispatchError>;
	// ...
```

Listing 4.15: Project: Trait definition part 1

```
        // As a participant, vote to replace a colleague
        fn vote_replace(pl: Self::IdentityId, worker: Self::IdentityId, project: ProjectID)
            -> Result<(), DispatchError>;
        /// Get project
        fn get_project(project: ProjectID)
            -> Result<Option<Self::Project>, DispatchError>;
        /// Get all projects
        fn get_projects() -> Result<Vec<Self::Project>, DispatchError>;
}
```

Listing 4.16: Project: Trait definition part 2

The trait definition demands for the implementation of functions to spawn a project, apply as a project leader, vote for a project leader, open a position as a project leader, apply as a worker to an open position, offer an applicant a role and a salary, accept a role as a worker, vote to replace a colleague and to retrieve information about a project. The trait uses the associate type `Balance` to know the currency type that is used in the system to payout rewards. The `IdentityId` type is required because the project needs to know to which type the data of a participant should be associated to. Like in the Council pallet, any type that has the same properties as the intrinsic address type can be used. In this case, the type is provided by the Identity pallet. The `ProposalWinner` type is used to determine how the information is structured, from which a new project is spawned. In this system, the type is provided by the Proposal pallet. The `Project` type contains all the necessary information about a project. In this case, the Project pallet provides this type. It is described in section 4.6.2.

4.6.2. Types

```
pub struct Worker<Balance, BlockNumber, IdentityId> where
        Balance: Codec + Clone + Debug + Eq + PartialEq,
        BlockNumber: Codec + Clone + Debug + Eq + PartialEq,
        IdentityId: Codec + Clone + Debug + Eq + EncodeLike,
{
        worker: IdentityId,
        job_description: DocumentCID,
        salary: Balance,
        hired: BlockNumber,
}
```

Listing 4.17: Project: Worker type

The Project pallet offers two custom types: `Worker` (listing 4.17) and `Project` (listing 4.18). The Worker type contains all the relevant information about a worker, including the project leader. The `Worker` type is contained within the `Project` type, which contains all information about one specific project.

```
pub struct Project<Balance, BlockNumber, IdentityId> where
        Balance:    Codec + Clone + Debug + Eq + PartialEq,
        BlockNumber: Codec + Clone + Debug + Eq + PartialEq,
        IdentityId:     Codec + Clone + Debug + Eq + EncodeLike,
{
        pub id: ProjectID,
        pub proposal: ProposalWinner<IdentityId>,
        pub project_leader: Option<Worker<Balance, BlockNumber, IdentityId>>,
        pub open_positions: Vec<DocumentCID>,
        pub workers: Vec<Worker<Balance, BlockNumber, IdentityId>>,
        pub deadline: BlockNumber,
}
```

Listing 4.18: Project: Project type

4.7. Proposal

The Proposal pallet offers a complete implementation of a proposal system as described in the design chapter. The definition of the Trait trait, storage, events, errors and the module logic is contained within the file at `implementation/superorganism/pallets/proposal/lib.rs`. Custom types of the Proposal pallet are defined in `implementation/superorganism/pallets/proposal_types/lib.rs`. Usually the custom types are in the same folder as the source file that contains the logic, in this case a circular dependency between the Proposal pallet and the Project pallet enforced a separation of the logic and the types into two different projects.

4.7.1. Configuration

The configuration of the pallet is described in this section and split into two categories: Constants and dependencies.

Constants

```
pub const IdentifiedUserPenality: u32 = TwoYears::get();
pub const ProposeCap: u32 = 1_000;
pub const ProposeIdentifiedUserCap: u8 =  1;
pub const ProposeIdentityLevel: u8 = 2;
pub const ProposeReward: Permill = Permill::from_percent(5);
pub const ProposeRoundDuration: BlockNumber = OneWeek::get();
```

Listing 4.19: Proposal: Configuration values for the Propose state

The configuration is logically separated into five sections, which correspond to each of the five states. The first section defines constants that are relevant in the Propose state, during which DDIs can submit proposals. Listing 4.19 shows those configuration values. `IdentifiedUserPenality` is the only value that is not yet integrated into the proposal pallet. The idea behind this value is to specify how long a DDI should be locked out for malicious behavior, for example copying other proposals and applying minor changes, like removing one letter. As described in the design chapter, only CIDs are stored as a memory pointer to an off-chain storage, because everything stored in the blockchain is replicated on every full node. Since the CID of a document changes completely when the contents are modified, because it is based on a cryptographic hash function, minor changes in the document result in a completely different CID. The proposal pallet can only verify if a CID of a proposal already exists in a system, because verifying the contents of a document is not deterministic in terms of duration it takes to fetch the document and in terms of persistence. In a future version of this system, users should help identifying users that copy other proposals and flag them. The `IdentifiedUserPenality` is the duration a user is locked out from proposing proposal for such behavior.
The `ProposeCap` is the number of new proposals the system can manage during the Propose state, which is set to 1,000 here. In terms of storage, the system can manage many more than 1,000 proposals. For example, if a CID would consist of only 32 Bytes, 1,000,000 proposals would require a storage capacity of 32MB. This is maintainable. The problem arises during the computation time required to evaluate them. Every proposal is an element within the storage that must be fetched one by one. As the time of writing this work, no optimization for bulk database requests have been implemented yet. As stated in the source code of the frame_support pallet, which offers weights for database accesses, the read time required for one storage element is about 25µs using the default RocksDB or 8µs using the experimental ParityDB. When fetching 1,000 proposals, the default database needs approximately $25\mu s \cdot 1,000 = 25ms$ to fetch all

proposals. This is manageable. In contrast to this, fetching 1,000,000 proposals would take approximately $25\mu s \cdot 1,000,000 = 25s$. This is by far not manageable. Given that the desired block time is about 6 seconds, just fetching the proposals from the database would take about four times the desired block time.
In addition to a maximum number of proposals, a maximum number of proposals per DDI is configured through the `ProposeIdentifiedUserCap` constant. If the number of proposals per user would be unlimited, a small group of users, even possibly a single user, could propose all proposals. `ProposeIdentityLevel` sets the minimum DDI level required to submit a proposal. `ProposeReward` sets the percentaged reward for a proposal that is converted to a project. Since the total reserved coins for a proposal are only known after an associated project has been spawned and a project leader has been selected, it is not used in the Proposal pallet. The `ProposeRoundDuration` constant sets the duration of the Propose state, during which DDIs with a level of `ProposeIdentityLevel` or higher may publish `ProposeIdentifiedUserCap` proposals.

```
pub const ProposeVoteAcceptanceMin: Permill = Permill::from_percent(10);
pub const ProposeVoteDuration: BlockNumber = OneWeek::get();
pub const ProposeVoteIdentityLevel: u8 = 3;
pub const ProposeVoteMaxPerIdentifiedUser: u16 = 3;
pub const ProposeVoteCorrectReward: Balance = 100_000_000_000;
```

Listing 4.20: Proposal: Configuration values for the VotePropose state

The second section defines constants that are relevant in the VotePropose state, during which DDIs can vote for proposals that were submitted during the previous state. Listing 4.20 shows those configuration values. `ProposeVoteAcceptanceMin` defines, how many percent of all votes for proposals are required at the end of the VotePropose state such that a proposal is passed to the next state. Usually a percentage would be assigned to a variable of the float (`f32`) or double (`f64`) type, but in this case those types are not applicable. This is due to the circumstance that those types may have different values depending on the architecture they are calculated on. The calculations within the runtime are verified by every full node in the ecosystem, therefore by using floats, the participants may come to conclusion that a block is invalid because the floating point numbers they calculated do not match with the floating point numbers stored (indirectly via a merkle root) in the block. Therefore the `Permill` type is used, which represent a float in the range of $[0, 1]$ as a fixed-point fractional number.
`ProposeVoteDuration` is used to specify the length of the voting period. `ProposeVote-`

`IdentityLevel` and `ProposeVoteMaxPerIdentifiedUser` define the DDI level required to vote and how many times a DDI may vote. `ProposeVoteCorrectReward` is the reward that is paid out to users, who cast a vote for a proposal whose vote percentage is greater than `ProposeVoteAcceptanceMin` at the end of the VotePropose state. In this case, the reward is an absolute value that is measured in units, while one unit is the smallest possible value of the cryptographic currency used in the system. Since this value is absolute, an additional mechanism is required to adjust the number of units depending on the price of the cryptographic currency on the market. Substrate offers a solution to query HTTP APIs and include the data once it is available. The reward adjustment mechanism is not part of the implementation but should be regarded in future development of this pallet.

```
pub const ConcernCap: u32 = 1_000;
pub const ConcernIdentifiedUserCap: u8 = 1;
pub const ConcernIdentityLevel: u8 = 2;
pub const ConcernReward: Balance = 10_000_000_000_000;
pub const ConcernRoundDuration: BlockNumber = OneWeek::get();
```

Listing 4.21: Proposal: Configuration values for the Concern state

The third section defines constants that are relevant in the Concern state, during which DDIs can submit their concerns about proposals that were passed into this state. Listing 4.21 shows those configuration values. In summary, all those configuration values are the same like those of the Proposal state, but they apply to concerns instead of proposals. The only difference is that the rewards, that are paid out to users for submitting a concern that is accepted (by receiving enough votes during the next state), is measured in units instead of a percentage.

```
pub const ConcernVoteAcceptanceMin: Permill = Permill::from_percent(3);
pub const ConcernVoteDuration: BlockNumber = OneWeek::get();
pub const ConcernVoteIdentityLevel: u8 = 3;
pub const ConcernVoteMaxPerIdentifiedUser: u16 = 3;
pub const ConcernVoteCorrectReward: Balance = 10_000_000_000;
```

Listing 4.22: Proposal: Configuration values for the VoteConcern state

The fourth section defines constants that are relevant in the VoteConcern state, during which DDIs can submit their votes for concerns that were passed into this state.

Listing 4.22 shows those configuration values. In summary, all those configuration values are the same like those of the VoteProposal state, but they apply to votes for concerns instead of votes for proposals.

```
1 pub const CouncilVoteRoundDuration: BlockNumber = OneWeek::get();
2 pub const CouncilAcceptConcernMinVotes: Permill = Permill::from_percent(85);
```

Listing 4.23: Proposal: Configuration values for the VoteCouncil state

The fifth and last section defines constants that are relevant in the VoteCouncil state, during which members of the council submit their votes for or against the realization of a proposal. Listing 4.23 shows those configuration values. `CouncilVoteRoundDuration` defines how much time the council has to cast votes for or against the realization of a proposal. `CouncilAcceptConcernMinVotes` is used to configure how many percent of the council members must have voted for a concern being too serious to accept the associated proposal, which ultimately leads to a refusal of the proposal.

Incorporated pallets

A few pallets have been loosely incorporated, as shown in listing 4.24. The issuance of new coins is controlled by the Balances pallets (line 1). The Scheduler pallet is used to schedule a state transition (line 2), as described in section 4.7.3. The Identity pallet is used to associate the address of a user, who calls a function in the pallet via a signed transaction, with a DDI. It is also used to retrieve the DDI level of the user. The council pallet is used to initiate a poll about the realization of winning proposals and their corresponding concerns and to fetch the results (line 4). The Project pallet is used to spawn a new project from a proposal (line 5), in case it passed all 5 states.

```
1 type Currency = pallet_balances::Module<Runtime>;
2 type Scheduler = pallet_scheduler::Module<Runtime>;
3 type Identity = pallet_community_identity::Module<Runtime>;
4 type Council = pallet_council::Module<Runtime>;
5 type Project = pallet_project::Module<Runtime>;
```

Listing 4.24: Proposal: Loosely incorporated pallets

4.7.2. State-Machine

The pallet implements a state machine as described in the design chapter in section 3.5. The states are declared in an enumeration type within the `proposal_types` package

and are shown in listing 4.25. The `#[derive(...)]` attribute instructs the compiler to derive an implementation for the traits specified within the round brackets after the `derive` keyword. For example, since the trait `Eq` is derived, variables that have the `States` type can be compared with each other by using the comparison operator. The `Uninitialized` state was not explicitly mentioned in the design chapter, but is depicted as the black dot in figure 3.9

```
#[derive(Copy, Clone, Debug, Decode, Encode, Eq, PartialEq)]
#[cfg_attr(feature = "std", derive(Serialize,   Deserialize))]
pub enum States {
        Uninitialized,
        Propose,
        VotePropose,
        Concern,
        VoteConcern,
        VoteCouncil,
}
```

Listing 4.25: Proposal: Enumeration type States containing possible states

4.7.3. Starting the state machine

When a program that is controlled by state machine is started as a standalone application on a single computer, the state machine is started during the startup of the program. In this scenario, the state machine is run as a distributed synchronized state machine over potentially thousands and more computers. The synchronization happens with the help of the consensus mechanism, that allows the participants in the distributed network to achieve consensus about the next state of the system. When is the program (in this case the Proposal pallet) initialized? There are two options. The first option is when the pallet was available in the runtime from the very first block, the genesis block, onward. Another option is that the pallet was added via a forkless runtime upgrade.

Starting the state machine at genesis

Executing a function during genesis that invokes another pallets (in this case Scheduler) function is cumbersome in Substrate. Substrate offers special functions in the module definition of a pallet that are executed every time a block is initialized or finalized. One option is to check if the state of the state machine is `Uninitialized` during block initialization, in that case the state machine is started. This has the enormous disadvantage, that the module is called every time a block is initialized and checks the

state of the pallet although it is required only once throughout its complete lifetime. Another variant that is not as trivial as the first variant is to invoke a function during the initialization of the pallets storage. This is the only entry point the runtime can call in a pallet during genesis in Substrate v2.0.0. As a matter of fact, this is abusing a functionality that the runtime offers, because what really happens is that during the initialization of the Proposal pallets storage a function is called, that calls the Scheduler pallet which in turn modifies its own storage. Listing 4.26 shows the code that leads to the described effects.

```
1  decl_storage! {
2      trait Store for Module<T: Trait> as Proposal {
3          pub State get(fn state) config(): States = States::Uninitialized
4          // ...
5      }
6      add_extra_genesis {
7          build(|_| {
8              let _ = <Module<T>>::do_state_transit();
9          });
10     }
11 }
12
13 impl<T: Trait> Module<T> {
14     fn do_state_transit() -> DispatchResult {
15         // ...
16         match state {
17             States::Uninitialized => {
18                 *state = States::Propose;
19                 transit_time = T::ProposeRoundDuration::get();
20             },
21             // ...
22             if T::Scheduler::schedule(...)
23             // ...
24         }
25     // ...
26     }
27 }
```

Listing 4.26: Proposal: Starting the state machine during storage initialization at genesis

The first strange part of that code is that the `State` storage indicated that it needs to be initialized by using the `config()` option in line 3, yet the storage is already initialized with the correct default value `States::Uninitialized`. It is required that at least one storage entry is configured, so that the function `add_extra_genesis` is

invoked, which does the actual work of initializing the state. For this reason, the genesis configuration at `implementation/superorganism/node/source/chain_spec.rs` must be appended by a value that would be used in a closure (anonymous function) within the `config()` function call. Listing 4.27 shows the extension of the genesis configuration by a dummy value that is in fact never used. Having made the runtime believe that the pallet wants to initialize its storage, the special function `add_extra_genesis` now will be invoked by the runtime at genesis. As shown in listing 4.26 in line 8, the modules `do_state_transit` function is called, which checks and modifies the current state and finally calls the Scheduler pallet with arguments that specify when the `do_state_transit` function should be called again, as shown in lines 16-22 of listing 4.26.

```
1 fn testnet_genesis(...) -> GenesisConfig {
2     GenesisConfig {
3         // ...
4         pallet_proposal: Some(ProposalConfig {
5             state: Default::default()
6         }),
7     }
8 }
```

Listing 4.27: Proposal: Adding a dummy value to the genesis config

Starting the state machine after a forkless runtime upgrade

The runtime offers a special function for a pallets module definition, `on_runtime_upgrade`, which is invoked after a new runtime was included in a block. Unfortunately, this function is called when the state of the runtime is not initialized and therefore the current block number cannot be retrieved. This is critical, because the Scheduler pallet demands for a block number as the future point of time a call should be scheduled at. Therefore it is impossible to schedule a call using the Scheduler pallet within the `on_runtime_upgrade` function at the time this work was created. Another variant that always works was described previously. By using the special function within the module that is invoked at the initialization of a block, the Proposal pallet can check whether its state was initialized yet and initialize it in case it was not done yet. As described previously, this is an inefficient method.

4.7.4. Dispatchable functions

The Proposal pallet offers five dispatchable functions, that can either be called by a module within the runtime (including pallets) or as an effect of the execution of an

extrinsic. Those dispatchable functions are defined within the `decl_module!` macro and are namely:

- `state_transit(origin)`
- `propose(origin, proposal: ProposalCID)`
- `vote_proposal(origin, proposal: ProposalCID)`
- `concern(origin, concern: ConcernCID, proposal: ProposalCID)`
- `vote_concern(origin, concern: ConcernCID, proposal: ProposalCID)`

In any case, the function parameter `origin` contains the information where the call originated from. The `state_transit` function can only be invoked by another module and potentially leads to a transition of the state. During the transition of the state, the storage is updated accordingly and relevant proposals or concerns are transferred into the next state. The four other functions listed can only be invoked by a signed transaction. All those four functions can only be successfully executed, if the caller owns a DDI with an appropriate level, as specified in the configuration. `propose` can be called during the Propose state and stores the proposal CID stored in the parameter `proposal` in the storage of the pallet and associates it with the DDI who called the function. `vote_proposal` can be called during the VotePropose state and in case the caller can still cast votes and the proposal specified in the parameter `proposal` exists, a vote is added to the proposal. `concern` can only be called during the Concern state and it enables a user to submit a concern for a specific proposal. `vote_concern` can only be called during the VoteConcern state and enables a users to submit a vote for a specific concern and proposal pair.
When the `state_transit` function is called by a valid origin (which only should be the Scheduler pallet), the evaluation of the data collected during the current state begins. In the implementation of the Module struct, many helper functions exists that the `state_transit` function uses to process the data collected during the state. For example, those helper functions help finding the proposals and concerns that have a voting ratio that is high enough.
Every dispatchable function has the same structure: It ensures first that all preconditions are met before making any changes to the state. This is a recommended approach called “verify first, write last”. If the preconditions are met, another function is called which actually processes the request.
Listing 4.28 shows the dispatchable function `propose`, which will serve as an example to explain the concept. The `ensure_signed` macro is used to ascertain, that the function call originated from a signed transaction. If the condition is not met, a predefined error is returned. After that, many invocations of the `ensure!` macro follow.

```
1  /// As an identified user, submit a proposal
2  #[weight = 10_000 + T::DbWeight::get().reads_writes(6,3)]
3  fn propose(origin, proposal: ProposalCID) {
4      let caller = ensure_signed(origin)?;
5      // Ensure that the pallet is in the appropriate state
6      ensure!(<State>::get() == States::Propose, Error::<T>::WrongState);
7      // Ensure that the maximum proposal count was not reached yet
8      ensure!(<ProposalCount>::get() < T::ProposeCap::get().into(),
9                  Error::<T>::ProposalLimitReached);
10     // Ensure the identity level is high enough to propose.
11     let id: IdentityId<T> = T::Identity::get_identity_id(&caller);
12     ensure!(T::Identity::get_identity_level(&id) >= T::ProposeIdentityLevel::get().into(),
13                 Error::<T>::IdentityLevelTooLow
14     );
15     // Ensure the user has not surpassed the proposal limit per user
16     ensure!(<Proposals<T>>::get(&id).len() < T::ProposeIdentifiedUserCap::get().into(),
17                 Error::<T>::UserProposalLimitReached
18     );
19     // Ensure that the proposal was not already submitted
20     ensure!(<ProposalToIdentity<T>>::get(&proposal) == IdentityId::<T>::default(),
21                 Error::<T>::ProposalAlreadySubmitted
22     );
23     Self::add_proposal(id, proposal);
24 }
```

Listing 4.28: Proposal: function `propose` as an example for the "verify first, write last" concept

The `ensure!` macro takes two arguments and allows the developer to couple a precondition to an error. The first argument is the condition, the second argument is the error that is returned in case the condition was not met. The errors can be custom errors, which were previously declared by using the `decl_error!` macro. Note that the return type of the function is implicitly added by the macro. The code in the listing 4.28 is well documented by comments, therefore the preconditions are not further explained. It is noteworthy though, that the storage is used extensively and compared to preconditions defined in the configuration section of the pallet and that custom errors are returned in case the preconditions are not met. Until the last expression in the function (line 22) is evaluated, no changes to the state were applied. In this scenario, a proposal is only added if all preconditions are met. The actual inclusion of the proposal into the storage of the pallet happens in a separate function, `add_proposal`, that is implemented in the implementation block of the `Module` struct, as shown in listing 4.29.

```
/// Add proposal to storage and update relevant storage values
fn add_proposal(id: IdentityId<T>, proposal: ProposalCID) {
        //   Create proper Proposal and add it to the users list   of proposals
        let    document = Proposal::new(proposal.clone());
        <Proposals<T>>::mutate(&id, |user_proposals| {
                user_proposals.push(document);
        });
        //   Add mapping from proposalCID to identity
        ProposalToIdentity::<T>::insert(&proposal,   &id);
        //   Increment total proposal count
        <ProposalCount>::mutate(|pc| *pc += 1);
}
```

Listing 4.29: Proposal: Modification of storage after all preconditions are met

4.8. Interaction with the system

To build the Substrate node and the frontend, an internet connection is required. The Substrate node is located at `implementation/superorganism`, whereas the frontend is located at `implementation/superorganism-frontend`. The frontend is based on the Polkadot-js/apps[1] project and was not developed by the author of this work, it was only configured to properly interact with the custom Substrate node that was designed and implemented during this work. The instructions given in the following section only apply to a Linux system.

4.8.1. Building and starting the Substrate node

In the folder containing the Substrate node, the following command is used to build the blockchain: `cargo build`. To build an optimized version, the command `cargo build --release` is used, but the compilation time increases due to the optimizations the compiler conducts. Once the Substrate node was built, the system can be started by using the command `cargo run -- --dev --tmp -lruntime=debug`. By omitting the `--tmp` flag, the chain state is stored on the hard drive and persists after the node is shut down. Building the runtime requires a web-assembly target for the rust compiler. When using the tool rustup to manage rust versions, the following command is sufficient to add the target: `rustup target add wasm32-unknown-unknown --toolchain nightly`. Once the Substrate node is started, the frontend can be used to interact with it, as described in section 4.8.2

[1] https://github.com/polkadot-js/apps

4.8.2. Preparing, starting and accessing the frontend

To build and run the frontend, Node (version $\geq$ 10.13.0) and Yarn (version $\geq$ 1.10.1) are required. In the folder containing the frontend, the command `yarn` can be used to prepare the project. The subsequent execution of the command `yarn run start` starts the frontend. Although the command line prints that the web server is ready, the frontend is not completely built and ready until the last message in the terminal ends with "HtmlWebpackPlugin_0 [built]".
Once the frontend is ready, a browser can be used to interact with it at the address "`localhost:3000`". The frontend can be used to submit extrinsics, for example to submit a proposal or a vote. It can also be used to read the storage, inspect blocks and extrinsics, events, account balances, etc.

4.9. Result

The Substrate node that was implemented during this work proofs that fundamentally, a decentralized morphogenetic system that is operated and formed by its community in a truly democratic fashion is possible. The implementation only provides interfaces, types and dummy implementations of the Identity pallet, Council pallet and Project pallet, therefore any further evaluation should focus on implementing those pallets. The Proposal pallet is fully functional and the implementation shows, that it can provide the community with a decentralized mechanism to share their ideas and concerns and that it can distribute rewards for participating in selecting the ideas and their associated concerns that will be converted to a project. The pallet is able to achieve this reliably while only using an acceptable fraction of the total computation time available for a block. Unfortunately, the OffchainWorker pallet is missing in this implementation, therefore the reward-based inclusion of a decentralized storage and computation system that is not part of the runtime has yet to be implemented and verified.

5. Conclusion

A design was created and implemented in this work that uses the Substrate blockchain framework as a modular dipt-blockchain. The design proposes a new concept called a morphogenetic blockchain, that describes how a dipt-blockchain develops its default features over time by the will of the community. The community chooses the features collectively, implements them and operates the resulting features. This is achieved in a truly democratic fashion, that prevents central authorities and might through wealth by design. A potential solution for a long-lasting difficult problem was described, that serves as a basis of the whole design and especially the truly democratic nature of the system: The validation of a human being without central authorities, such that every human being at most can own exactly one identity in the system. The identity solution is used in the other modules, that support the system with the functionality to:

- let the community elect a council, which is in full control of the community and which serves the community by resolving difficult problems
- let the community propose ideas and concerns about the next features the system should support, while being rewarded for correct decisions
- convert the proposals and concerns into a project and manage the creation, realization and finalization of the project

In addition, a rough idea was presented, that suggests to outsource storage and computation requirements to separate decentralized applications that are only linked to the blockchain by unique identifiers. A solution how to bootstrap the system with the initial NPoS validators and authorities and how to bootstrap the system with initial DDIs was also presented.
The implementation supplied with this work realizes the design that was described partially, with exception of offchain storage and computation. The identity, council and project designs were implemented using interface definitions, custom types and dummy implementations. A complete functional implementation of their logic is still necessary. Based on those dummy implementations, the proposal design was completely implemented. It shows, that a complete truly democratic proposal and vote process to

determine the next features of the dipt-blockchain, while rewarding the participants for correct decisions, is possible.
Future steps that are required to complete the system are (ordered by priority):

1. Design and PoC-implementation of a fully functional offchain storage and computation solution that is linked to the blockchain by identifiers and appropriately rewards storage and computation time suppliers.

2. Creation of a more detailed definition for the project management system.

3. Complete implementation of the DDI management system, based on the interface definition that was supplied with this work. Implementation of the offchain meeting and verification module.

4. Extension of the project management interface, based on the more detailed definition and an audit solution. Complete implementation of the project management system based on the extended interface.

5. Complete implementation of the council logic.

When those tasks were successfully completed, the system can be fine-tuned. This includes the transfer from a PoS system to a novel PoI system, that uses the novel feature that every human can only have one DDI in its consensus rules. For example, instead of staking the intrinsic currency for a participation right in the block authoring process, the own identity could be the stake. Meaning that in case the DDI acts maliciously, it is locked out from the block authoring process for a long time. If the DDI acts righteously, it is rewarded with coins. This type of consensus mechanism is majorly unexplored, mainly due to the fact that no proven solutions exists yet that ensures that every human being only has maximally one DDI in the system. In addition to the adjustment of the consensus mechanism, the fee-based transaction concept should be revised. Whenever possible, a DDI should be able to participate for free. For example, in the system presented in this work, every DDI can submit 1 proposal per proposal round (which should take about one week or longer). If the DDI used up their limit, their transaction should be rejected. This concept should be applied whenever possible, to remove the requirement to own a currency to be able to participate in democratic processes. At last, it might be helpful to integrate the knowledge of Sociocracy 3.0[BPD], a social technology for evolving agile and resilient organizations at any size, into the design of the system. Much practical knowledge about how an organization can be optimally managed, while keeping flat hierarchies and integrating every opinion in the decisive process of current and future projects of the organization, lead to a guideline that probably can be applied to a system like the system described in this work.

This project is vast, in fact it is too much for one thesis, it is even too much for one faculty. This is a master thesis for the computer science degree, therefore the technical possibilities can be evaluated, yet this project is interdisciplinary. For example, it has its own currency that is issued at different occasions. Economists are required to evaluate the economic plausibility of the decisions made in this work, and in addition should optimize and propose a better solution. This project also incorporates a lot of game theory, for example which roles the participants fulfill and which move every participant in the democratic game can execute at different states of the system. The participants in the system define the systems features and should be motivated to get the best out of it. Psychologists have a better understanding of how an individual perceives decisions in the system. Those are only some example, certainly more disciplines are required. In summary, an interdisciplinary study about a system like the system proposed in this work is required, such that the system matures and is able to last over a longer period of time, while being of use to the community that operates it.

A. Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
AURA	Authority-Round
BABE	Blind Assignment for Blockchain Extension
CID	Content IDentifier
CTP	Computation Time Provider
dipt	decentralized, immutable, permissionless and trustless
DDI	Decentralized Digital Identity
DHT	Distributed Hash Table
DLT	Distributed Ledger Technology
DOS	Denial of Service
DPoS	Delegated Proof-of-Stake
FAR	False Acceptance Rate
FNIR	False Negative Identification Rate
FPIR	False Positive Identification Rate
FRAME	Framework for Runtime Aggregation of Modularized Entities
FRR	False Rejection Rate
GRANDPA	GHOST-based Recursive Ancestor Deriving Prefix Agreement

Abbreviation	Meaning
IPFS	InterPlanetary File System
Merkle DAG	Merkle-Directed Acyclic Graph
NPoS	Nominated Proof-of-Stake
PoC	Proof-of-Concept
PoI	Proof-of-Identity
PoS	Proof-of-Stake
PoW	Proof-of-Work
LOC	Lines of code
SSI	Self-Sovereign Identity
STF	State Transition Function
TXID	Transaction ID
UTXO	Unspent Transaction Output
VRF	Verifiable Random Function
W3C	World Wide Web Consortium
Wasm	WebAssembly

List of Figures

Listings

Bibliography

[Alp20] Handan Kılınç Alper. *BABE*. https://research.web3.foundation/en/latest/polkadot/block-production/Babe.html. Accessed: 2020-25-11. 2020.

[Ben14] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *CoRR* abs/1407.3561 (2014). arXiv: 1407.3561. URL: http://arxiv.org/abs/1407.3561.

[BH07] Mario Cortina Borja and John Haigh. "The birthday problem". In: *Significance* 4.3 (2007), pp. 124–127. DOI: 10.1111/j.1740-9713.2007.00246.x.

[BMZ18] L. M. Bach, B. Mihaljevic, and M. Zagar. "Comparative analysis of blockchain consensus algorithms". In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2018, pp. 1545–1550. DOI: 10.23919/MIPRO.2018.8400278.

[BPD] Bernhard Bockelbrink, James Priest, and Liliana David. *A Practical Guide for Evolving Agile and Resilient Organizations with Sociocracy 3.0*. https://sociocracy30.org/_res/practical-guide/S3-practical-guide.pdf. Accessed: 2020-12-05.

[But20] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Tech. rep. Ethereum, Oct. 2020. URL: https://ethereum.org/en/whitepaper/.

[Cam04] J. L. Camp. "Digital identity". In: *IEEE Technology and Society Magazine* 23.3 (2004), pp. 34–41. DOI: 10.1109/MTAS.2004.1337889.

[Chu+20] Mengyu Chu et al. "Learning Temporal Coherence via Self-Supervision for GAN-based Video Generation (TecoGAN)". In: *ACM Transactions on Graphics (TOG)* 39.4 (2020).

[CS20] Alfonso Cevallos and Alistair Stewart. *A verifiably secure and proportional committee election rule*. 2020. arXiv: 2004.12990 [cs.DS].

[Fra18] Jack Fransham. *What is Substrate?* 2018. URL: https://www.parity.io/what-is-substrate.

[GH19] H. Gulati and C. Huang. “Self-Sovereign Dynamic Digital Identities based on Blockchain Technology”. In: *2019 SoutheastCon*. 2019, pp. 1–6. DOI: 10.1109/SoutheastCon42311.2019.9020518.

[Gmb20] BOTLabs GmbH. *KILT*. Tech. rep. 10787 Berlin, Germany: BOTLabs GmbH, Jan. 2020. URL: https://www.kilt.io/wp-content/uploads/2020/01/KILT-White-Paper-v2020-Jan-15.pdf.

[GNH20] Patrick Grother, Mei Ngan, and Kayee Hanaoka. *Face Recognition Vendor Test (FRVT)*. Tech. rep. Gaithersburg, Maryland: National Institute of Standards and Technology, Mar. 2020. URL: https://pages.nist.gov/frvt/reports/1N/frvt_1N_report.pdf.

[Gup15] Ganesh Gupta. *What is Birthday attack??* Feb. 2015. DOI: 10.13140/2.1.4915.7443.

[Hec20] Harald Heckmann. *Decentralized Identities*. https://medium.com/@harald.heckmann.mbw/decentralized-identities-4b93e05a35ff. Accessed: 2020-21-11. 2020.

[JL17] S. Josefsson and Ilari Liusvaara. “Edwards-Curve Digital Signature Algorithm (EdDSA)”. In: *RFC* 8032 (2017), pp. 1–60.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA)”. In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[KN12] Sunny King and Scott Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. Tech. rep. Decred, Aug. 2012. URL: https://decred.org/research/king2012.pdf.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* (1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/.

[Mer88] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.

[Mor68] Donald R. Morrison. “PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric”. In: *J. ACM* 15.4 (Oct. 1968), pp. 514–534. ISSN: 0004-5411. DOI: 10.1145/321479.321481. URL: https://doi.org/10.1145/321479.321481.

[MRV99] S. Micali, M. Rabin, and S. Vadhan. "Verifiable random functions". In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 120–130. DOI: 10.1109/SFFCS.1999.814584.

[Müh+18] Alexander Mühle et al. "A survey on essential components of a self-sovereign identity". In: *Computer Science Review* 30 (2018), pp. 80–86. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2018.10.002. URL: http://www.sciencedirect.com/science/article/pii/S1574013718301217.

[Nak09] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Bitcoin, 2009. URL: http://bitcoin.org/bitcoin.pdf.

[NJ20] N. Naik and P. Jenkins. "Self-Sovereign Identity Specifications: Govern Your Identity Through Your Digital Wallet using Blockchain Technology". In: *2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. 2020, pp. 90–95. DOI: 10.1109/MobileCloud48802.2020.00021.

[Ree+20] Drummond Reed et al. *Decentralized Identifiers (DIDs)*. 2020. URL: https://www.w3.org/TR/2020/WD-did-core-20200421.

[Rös+19] Andreas Rössler et al. "FaceForensics++: Learning to Detect Manipulated Facial Images". In: *ICCV 2019*. 2019, pp. 1–11. DOI: 10.1109/ICCV.2019.00009.

[SA15] Markku-Juhani Saarinen and Jean-Philippe Aumasson. Tech. rep. RFC 7693. Internet Engineering Task Force, Nov. 2015.

[Sch89] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *CRYPTO*. 1989.

[SK20] Alistair Stewart and Eleftherios Kokoris-Kogia. *GRANDPA: a Byzantine Finality Gadget*. 2020. arXiv: 2007.01560 [cs.DC].

[SLC19] Manu Sporny, Dave Longley, and David Chadwick. *Verifiable Credentials Data Model*. 2019. URL: https://www.w3.org/TR/2019/REC-vc-data-model-20191119/.

[Tol+20] Ruben Tolosana et al. *DeepFakes and Beyond: A Survey of Face Manipulation and Fake Detection*. 2020. arXiv: 2001.00179 [cs.CV].

[TV18] M. Takemiya and B. Vanieiev. "Sora Identity: Secure, Digital Identity on the Blockchain". In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 02. 2018, pp. 582–587. DOI: 10.1109/COMPSAC.2018.10299.

[Woo17] Gavin Wood. *Polkadot: A vision for a heterogenous multi-chain framework.* Tech. rep. Parity, 2017. URL: `https://polkadot.network/PolkaDotPaper.pdf`.

[Woo20] Gavin Wood. *Ethereum: A Secure decentralised generalised transaction ledger.* Tech. rep. Ethereum and Parity, Sept. 2020. URL: `http://ethereum.github.io/yellowpaper/paper.pdf`.

[Zha+20] Xuaner Zhang et al. "Portrait Shadow Manipulation". In: *ACM Transactions on Graphics (TOG)* 39.4 (2020).