energies

MDPI

Article

An Asynchronous AAA Blockchain-Based Protocol for Configuring Information Systems

Michał Bajor *,† and **Marcin Niemiec** †

Department of Telecommunications, AGH University of Science and Technology, Mickiewicza 30, 30-059 Krakow, Poland

* Correspondence: bajor.michal98@gmail.com

† These authors contributed equally to this work.

Abstract: The increasing number of security breaches in centralized systems provides the necessity to introduce decentralization in more fields. The Blockchain is a widely utilized decentralization technology that is implemented in various industries. Therefore, this technology can be used to protect sensitive services, such as those associated with the configuration changing of information systems. This article proposes a new protocol operating as a decentralization layer over any configuration scheme. It uses smart contracts—programs existing on the Blockchain—to keep track of configuration proposals and authorize new configurations. The configuration change can be proposed at any time. However, only once it is authorized by appropriate parties can it be introduced to the system. The new protocol provides an additional security layer, ensuring that every action is accounted for and authenticated. Furthermore, it enforces that administrators authorize every change. The protocol was designed to be flexible and easily adaptable to scenarios that did not use distributed ledger technology before. It uses the HTTP protocol with the JSON standard for protocol messages to allow easier adoption and transparency. The features of the proposed protocol were analyzed from a security point of view as well as from the financial perspective related to costs of using Blockchain technology. Security analysis shows that the protocol is resilient to the most common security risks that haunt state-of-the-art IT systems. Additionally, the authors proved that this solution could be implemented in both private and public Blockchains. A reference implementation was shared in a public repository. The proposed protocol was also compared with the most similar state-of-the-art work in the academic research highlighting the key differences and improvements.

Keywords: Blockchain; authorization; authentication; accounting; configuration; smart grids

Citation: Bajor, M.; Niemiec, M. An Asynchronous AAA Blockchain-Based Protocol for Configuring Information Systems. *Energies* **2022**, *15*, 6516. https://doi.org/10.3390/en15186516

Academic Editor: Joao Ferreira

Received: 29 July 2022
Accepted: 2 September 2022
Published: 6 September 2022

1. Introduction

It is safe to say that aspects of people's lives are now increasingly being digitized—social interactions, banking, and entertainment being the prime examples. However, due to network connectivity's benefits, increasing numbers of systems that previously worked offline are now connected to the network (often to the internet as well)—for instance, systems associated with production, fridges, cars, or in general, Internet of Things devices. However, there is also an increasing need for security in the underlying infrastructure, including power grids, which are now becoming smart grids, and in the energy sector in general [1–4]. Quite often, such information systems require precise configuration that will dictate how they should operate. The severity of such configuration differs from system to system. However, the proper configuration must be applied in a system because of security and efficiency. Moreover, important information is the time when the configuration was applied and who is responsible for this action. It is crucial to verify if a person responsible for changing the configuration had sufficient authority/privileges to do so. Many systems and devices require a strict change policy with comprehensive management. This paper proposes a structured approach to achieve a decentralized protocol for configuring such

systems using Blockchain technology. Furthermore, it is clear that standardization is required, as even the Ethereum platform decided to introduce standards for smart contracts in the form of Ethereum Request for Comments (ERC) [5].

The Blockchain is the most common implementation of a distributed ledger technology that tracks, processes, validates, and authenticates transactions. In essence, it is a decentralized database with no central administration. Decentralization eliminates a single point of failure. The defining characteristic of a Blockchain is its immutability. Once the transaction is recorded and published, it cannot be reversed or deleted. The first notable usage of Blockchain was introduced by Satoshi Nakamoto in [6], which is the foundation of Bitcoin—the most popular cryptocurrency. Bitcoin created a peer-to-peer network allowing a trustless exchange of value. It is often referred to as "Blockchain 1.0". Newer projects introduced the possibility of executing arbitrarily written code in Touring-complete programming languages. This on-chain program is most often called a smart contract and has its own address on the Blockchain once deployed. Blockchains supporting smart contracts are often referred to as "Blockchain 2.0". Currently, the community is shifting to "Blockchain 3.0", which is focused on integrating Blockchain 2.0 technology with more aspects of human life, not limiting it to the financial world.

1.1. Motivation

There is no denying that, currently, society is used to operating in a centralized model. For example, every financial operation, be it a money transfer, compensation for work, or even a purchase of commodities, needs to go through the third party that needs to accept the operation. Furthermore, there is a massive flood of fake news and targeted advertisement only because people use systems designed to generate the most profit for its creators—often big tech companies. There is nothing wrong with using such systems if one wishes to. However, it still requires users to trust that their data or actions are performed in a secure and privacy-preserving manner, and even that they will be performed at all.

One of the paradigms of decentralization is that users do not need to trust any third party to do anything. Instead, such security should be implemented by design and incentivized by the whole ecosystem instead of the good will of a particular system's owner. This is why Blockchain technology is gaining popularity and is being adapted into more domains. This paper attempts to introduce decentralization into one more field—change control in IT systems.

A high-level usage example of the protocol proposed in this paper is as follows. Let us suppose there is a factory producing equipment in a robotic production line. In such a critical scenario, multiple people from different departments must approve every change in the production parameters, including administration, management, and quality assurance. The proposed change may come from any of these departments with a different purpose. For instance, the management department might want to introduce a change in the production system to reduce operating costs. Such a change can be proposed at any time. However, after it is proposed, other departments need to review the change and authorize it only if they find it suitable. Once all required parties agree on the change, it will be introduced to the system. In this scenario, it is essential to keep track of who proposed and approved the specific change and when it happened.

1.2. Contribution

The protocol proposed in this paper uses Blockchain technology to facilitate the scenario described in the previous paragraph. Everyone wishing to suggest a change must record this on the Blockchain. Similarly, every authorization will also need to be performed on the Blockchain. This way, actions taken by interested parties are protected by asymmetric cryptography, and all of them are accounted as they are recorded on the public Blockchain. A significant part of the authorization is also transferred to the Blockchain as the smart contracts are used to keep track of proposals and authorizations.

Furthermore, the protocol defines a clear path that every change proposal should follow, which helps to create a transparent business process that is secure and easily verifiable. The proposed protocol was designed to be flexible and operate as an additional layer over the currently used configuration scheme. It can be used to configure any system. However, based on the features described in this paper, the protocol is best used in environments that require the highest possible level of accountability and transparency, such as factories or critical infrastructure, including smart grids. In summary, the contribution of this paper covers the following issues.

- Motivation and needs of decentralized solutions for configuring information systems are discussed.
- An innovative asynchronous configuration protocol based on Blockchain technology is introduced. The novel protocol is formally defined, including the commit structure, requests, responses, and smart contracts.
- Examples of lightweight objects and structures to support flexibility and applicability in modern communication networks are presented. Templates of smart contracts used to authorization purposes are provided.
- The functionality of the proposed protocol is verified, including analysis of Blockchain interaction fees and latency. Security considerations regarding the new protocol are provided.
- The new solution is implemented and shared in a public repository [7].

This article consists of six sections. Section 2 contains a description of the current state of the art. Section 3 introduces a high-level overview of the proposed protocol and describes a typical scenario. Section 4 includes defined requests and responses, as well as the description of smart contracts used in the protocol. The article is summarized in Sections 5 and 6.

2. State of the Art

Blockchain technology has taken the world by storm ever since a group or individual named Satoshi Nakamoto proposed Bitcoin in 2008 [6]. Initially, Blockchain technology was focused on purely financial aspect of society; however, as the technology evolved, it was introduced to more areas. Currently, it is also strongly applied for authentication, authorization, and accounting purposes. The research focuses on various aspects of Blockchain—one of them being connecting it with other technologies and looking for new applications [8,9]. There have been attempts at proposing a standardized approach for multiple applications, for instance, a decentralized authentication and authorization protocol similar to OAuth2 [10], an identity authorization mechanism based on two sub-chains with a registration process [11], and a protocol for configuring network devices based on Hyperledger [12].

As identified by Li et al. [9], the most recent Blockchain iteration (called Blockchain 3.0) concerns its integration with various industries. The community has identified that Blockchain can help with user authentication and authorization. For instance, Lin et al. [13] have developed a Conditional Privacy Preserving Authentication protocol for vehicular networks (VANET) using Blockchain, and specifically the Ethereum platform, which was proved to conform to security requirements along with providing a good user experience because of small delays. Perera et al., on the other hand, have created a Certificate Management Scheme using Blockchain structure for VANET [14]. The application of various Blockchains for data collection in the automotive industry was described in [15]. Abubakar et al. [16] created a lightweight authentication protocol for SIP-based VoIP systems also using the Ethereum Blockchain. Shahzad et al. have proposed an authentication solution based on Blockchain for communication security in 6G networks [17]. Blockchain is also used for supply chain security and product traceability [18–20]. Another industry benefiting from Blockchain technology is the Internet of Things, which uses it to facilitate data security and access control [21–23].

The general idea proposed by Helebrandt et al. [12] is similar to the one proposed in this paper, with some notable differences—primarily the fact that the protocol proposed in this paper is ready to be used in public Blockchains. The differences are stressed in the following sections. It is worth mentioning that some of the statements, although true at the time of publishing, are now outdated. Originally, the Blockchain technology was not associated with high throughput or scalability [8]; however, the world has now been introduced to Blockchains with a higher TPS (transactions per second—the value indicating how many transactions on average are processed by Blockchain) than Visa's [24]. The Solana Blockchain team, in fact, states that its scalability is limited only by current hardware.

3. A New Protocol

The idea behind using Blockchain technology to support an asynchronous configuration protocol providing authentication, authorization, and accounting (AAA) is not convoluted, especially when taking into account the fact that the Blockchain itself is a distributed ledger technology, and every interaction with it is public. Hence, the "accounting" part of a new protocol's functionality is ready out of the box.

The most common way to authenticate is a shared secret (passwords). This is easy to implement for developers, easy to maintain for administrators, and easy to use for users. However, one major drawback is a single point of failure—the database where hashes of users' passwords are stored. If a system using a shared secret for authentication is compromised, and such a database is leaked, usually, it is a matter of time before passwords are cracked using a brute-force approach. It might take a long time, but a determined attacker will crack them sooner or later. Moreover, such a scenario applies if the system was incorporating best practices and was not storing passwords in a clear-text format, which users can most often only hope for. Blockchain uses asymmetric encryption, so the equivalent of the password is a private key. However, it is not stored anywhere—it is known only to its owner. A private key is associated with a public key (known in the Blockchain world as an address). Hence, any interaction with Blockchain coming from a specific address is enough to know that this interaction was triggered by the person owning the private key associated with that address.

Authorization is a different concept. The most common authorization scheme is simply a field in a table storing users' identifications, which are assigned to specific groups with different privileges (for example, "staff", "guests", or "administrators"). Authorization is applied after successful authentication. Once the service knows who is trying to access or modify a given resource, it checks if that user has sufficient privileges using the mentioned table. For the proposed protocol to work, a Blockchain supporting smart contracts is required, so-called "Blockchains 2.0" or above (examples of such Blockchains are Ethereum or EOS; however, for example, Bitcoin does not meet the requirements). Smart contracts are essentially pieces of computer programs residing on the Blockchain that can be executed when certain conditions are met. Smart contracts are located directly on the Blockchain. Therefore, they are immutable—once defined and published, the smart contract will always work in the same manner. A smart contract associated with the proposed protocol needs to have a list of Blockchain addresses that identify administrators (or simply users with privileges required to perform a given action).

An asynchronous aspect of a new protocol is that anyone can propose a change at any given time, but they will be staged only once the administrator authorizes them. Contrary to the solution proposed in [12], the configured service, system, or device does not interact with Blockchain at any time—it is only notified which change should be staged based on what was authorized via smart contract. This approach makes the protocol more resistant to Denial of Service (DoS) attacks and is ready to be used both on the public and private Blockchains. The flow diagram of a new protocol is presented in Figure 1. It is worth mentioning the "server" in the figure represents a system's component responsible for applying the configuration. It might be a part of the configured system itself or a separate entity.

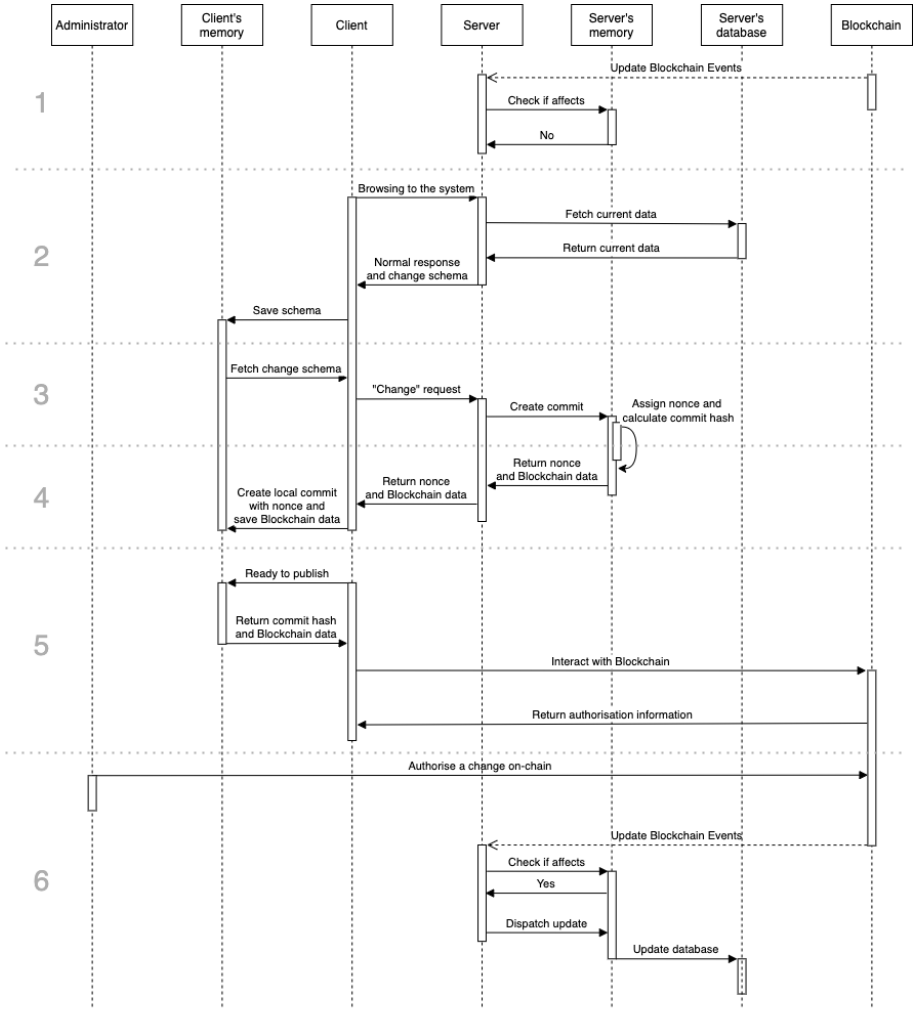

Figure 1. Flow diagram of the proposed protocol.

The essence of the protocol operation, based on the diagram seen in Figure 1, is the following list of actions (each number on the left side of the figure corresponds to the particular step):

1. The server continuously monitors the state of Blockchain looking for the emitted events. It will only act when events affecting that particular server are identified.
2. A client, which can be anyone, uses the system and receives the data currently saved in database along with the change scheme.
3. Then, once the client decides to propose a change, it sends a specific request using a previously saved schema.
4. The server creates a new commit associated with that change and assigns it a nonce, which is returned to the client along with Blockchain data (address and interface of the smart contract). At this point, no change was introduced and data was not changed.
5. Once the client decides to publish the change request, he or she interacts with the Blockchain via smart contract shared by the server. On successful interaction, authorization information is returned. It contains the address of a smart contract that the administrator needs to interact with to stage the changes.
6. Only once the administrator authorizes specific change is the server notified, and it updates the database with new data.

Let us assume the proposed changes are stored in a repository, for example, a database, and will be identified using the hash value of those changes. In order to introduce change into the system, it is necessary to interact with the smart contract associated with that par-

ticular system. If the change was proposed by an administrator (e.g., someone authorized to stage them), then the smart contract immediately emits an event (an event in terms of the Ethereum Smart Contract is in the form of a log) informing that changes identified by a particular hash can be staged. If additional authorization is needed, the smart contract will create another contract (called *Authorizer*) responsible for authorizing that particular change. Administrators who want to authorize this change interact with the *Authorizer* contract. This contract will inform the base contract that someone has authorized a change. Once the change is authorized via the Blockchain mechanism, the base contract will also emit an event informing that those changes can be applied. The server with access to a repository, where potential changes are stored, listens to those events. Once an event is received, the system validates and implements the associated changes.

The mechanism proposed in this paper is designed to be operational on a public Blockchain. Furthermore, the mechanism does not require a separate smart contract for each application (although it can be set up in such a manner). Contracts are agnostic when it comes to the configured underlining system, meaning that a single contract can operate as an endpoint for various services (as long as they share the same administrators).

4. Formal Protocol Description

This section describes the protocol in greater detail. It first focuses on the commit structure, which represents a specific change proposal. The following subsections describe requests and responses defined in the protocol. Then, this section is concluded with the characterization of smart contracts and a short note on governance in the protocol. The protocol was designed to be lightweight and easily adaptable to the existing solutions, hence the usage of a well-established and widely used JSON data format [25] for both data transmission and storage and HTTP protocol [26] used for communication. Furthermore, it was designed to be flexible and applicable in as many scenarios as possible.

4.1. Commit Structure

The solution proposed in this paper defines a structure of an update in configuration. This update is referred to as a "commit". The commit is a JSON object and consists of various fields, which are listed in Table 1.

Table 1. Fields present in the commit JSON object.

Field Name	Data Type
changes	JSON object
creator	String
nonce	Integer
additional_authorizers	JSON array
depends_on	JSON array

The keys of the *changes* JSON object are the variable names, which are dependent on the system under configuration. Values present in this object represent the change proposal. Should this commit be staged, the values present in the *changes* object would become the new configuration. This field is solely dependent on the application or system configured.

The string assigned to the *creator* key needs to contain the Blockchain address of the commit's creator, which is essentially his or her public key in hex representation prefixed with "0×". This element allows for the unambiguous identification of the creator.

The *nonce* is a 32-bit number randomly generated on the server-side for each commit. The nonce is supposed to increase the difficulty of a brute-force attack. Every interaction with the Blockchain is public. In order not to share the configuration publicly, all users refer to commits by their hashes. The protocol allows the situation where the configuration scheme is public, so anyone could know which parameters are configurable and what their

data types are. In such cases, an attacker would be able to generate hashes corresponding to different configuration proposals and act on them once spotted on the Blockchain. The creator's address is also part of the commit, influencing the commit's hash. However, the mechanism cannot rely on this. Although the protocol allows anyone to propose a change, in most cases, only specific people will be responsible for this. Hence, the *creator* key might be easily guessable. To prevent these brute-force attacks, the *nonce* field was introduced to the protocol so that no two commits share the same hash, even if all of their fields are the same.

The *additional_authorizers* JSON array is an array of strings, each representing a Blockchain address (in the same format as the *creator* key). This could be provided by the commit's initiator or filled in by the server or can even be stored in the smart contract itself—it would depend on the implementation. Regardless of which side is filling this information, if the *creator* is not authorized to stage changes (the creator is not a privileged administrator), then this field cannot be an empty array unless it is defined in the smart contract.

The *depends_on* JSON array is an array of strings. Each string is a hex representation of a hash of another commit. This list indicates that the commit holding this list cannot be staged (even if authorized by administrators) until all of the commits identified by their hashes in this list are also staged. This field is not optional. However, it can be an empty array, which indicates that this particular commit does not depend on any other. This list is filled by the creator.

As already mentioned in the *depends_on* description, the crucial part of this mechanism is a commit hash. It is calculated using the SHA3-512 hash function in the test implementation of this protocol, although this particular hash function used is not obligatory. It is, however, strongly encouraged not to use hash functions that produce output shorter than 256 bits. This hash is used as an identifier of a particular commit in the Blockchain, i.e., users refer to specific commits by their hash. Longer hashes decrease the chance of using a brute-force approach in guessing the commit's content, especially the value of the *changes* JSON object, which helps to keep the proposed configuration confidential.

An example of a commit is presented in Listing 1. This particular commit proposes to change an application's variable named "boolean_value" to a value of "true" and the "background_color" variable to a value of "red". It does not depend on any other commit and requires one administrator to authorize that change.

Listing 1. A commit example.

```
{
"changes": {
"boolean_value": true,
"background_color": "red"
},
"creator": "0xFFcf8FDEE72ac11b5c542428B35EEF5769C409f0",
"nonce": 98430,
"additional_authorizers: [
"0xb1E7bC63d4f537d746F1C1342D517963feB7C5C7"
],
"depends_on": [],
}
```

4.2. Requests

This subsection considers the structure of requests sent to a server, and the responses are described in the next subsection. The communication between a client and a server is based on HTTP protocol, as it is a well tested and widely used carrier protocol for various types of data. Every message sent from the client to server needs to have a specific structure, which is an HTTP POST request. The body of the request is a JSON object with two keys:

- *proto_action*, which is a string and is always necessary. It defines the type of request sent.
- *payload*, which is a JSON object and sometimes can be omitted (depended on the *proto_action*).

The *proto_action* defines what type of message is sent and what server should reply. The possibilities are described below.

- *get_change_from_hash* is used to get the human-readable commit based on its hash. This action needs interaction with the Blockchain, and it requires two separate requests to the server to complete. In the first request, the *payload* key of the request needs to contain a *commit_hash* key, which contains the hex representation of a commit's hash, and a *public_key* key which holds the Blockchain address of the user who wishes to see the proposed commit. The server is obliged to check if the *public_key* is associated with one of the administrators or creator of the commit and to check if the commit of this hash exists. If all of those are true, then the server generates a token along with a challenge associated with it so that it can be authenticated. It is important to be able to connect the token to the challenge associated with it. It might be stored in a repository on the server as an arbitrary connection. In the proof-of-concept implementation, the challenge was calculated as a hash of the token, which connects the challenge to the token. However, it does not allow the token to be obtained from the challenge. At this point, the server responds to a client. In the case of the second request, e.g., the token was authenticated via interaction with the smart contract, the client request instead of *public_key* needs to contain the *token* field with the authenticated token. The server must validate if the token was authenticated and associated with the correct commit. Exemplary requests can be found in Listings 2 and 3.

Listing 2. A *get_change_from_hash* initial request example.

```
{
"proto_action": "get_change_from_hash",
"payload": {
"commit_hash": "ac2be638082283e54...",
"public_key": "0xFFcf8FDEE72ac11b...",
}
}
```

Listing 3. A *get_change_from_hash* request with authenticated token example.

```
{
"proto_action": "get_change_from_hash",
"payload": {
"commit_hash": "ac2be638082283e54...",
"public_key": "0xFFcf8FDEE72ac11b...",
"token": "17fe335be2244c9d01e7c4f952a17019...",
}
}
```

- *get_config* does not require a *payload* key to be present. This is the message requesting the current configuration of the system. It is possible that the system should not share this configuration openly, in which case a similar mechanism as that for *get_change_from_hash* with one-time tokens and challenges can be implemented.
- *get_schema* instructs the server to return the schema of change. This includes the order of fields along with their data type. It informs the client about configurable parameters. Furthermore, it also dictates what the commit looks like, which allows a client to calculate the commit's hash.

- *change_config* is used to propose new commits. This is the core functionality of the protocol. It requires an interaction with the smart contract. The *payload* field, in this case, consists of a single key—*commit*. This key maps to the JSON object, which maps to the flattened *commit* structure described in Section 4.1, e.g., it consists of the same information. However, there are no nested JSON structures. An exemplary request is presented in Listing 4.

Listing 4. A *change_config* request example.

```
{
"proto_action": "change_config",
"payload": {
"boolean_data": true,
"background_color: "red",
"additional_authorizers": "0xFFcf...,0xE11B...",
"depends_on": "",
}
}
```

- *get_abi* is used to receive an ABI (ABI stands for Application Binary Interface—it is used in the Ethereum Blockchain to describe how to interact with the smart contract it is associated with) for either the base contract or authorizer contract. It requires a *payload* to be present in the request. Inside the *payload*, there needs to be an *abi_type* key mapping to either "base" or "authorizer".

4.3. Responses

The server response body is also a JSON object, which always contains a *success* key, which is a numeric value indicating a success (value 1 means successful operation, anything else is a failure). If the *success* value indicates a failure, then the response also needs to contain a *message* key, which will hold a string message describing the reason for failure. On the other hand, if the *success* value indicates successful operation, then other keys present in the response are dependent on the *proto_action*. Those responses are defined as follows.

- *get_change_from_hash*—after the initial client request, which results in token and challenge generation, the server returns a JSON object consisting of the *success* key set to 1, *token_value*, and *challenge* keys set to generated token and challenge, *contract_address* which holds the address of a smart contract to which the user must authenticate his or her token, and *abi*, which instructs the client how to interact with that smart contract. Response for the second request (i.e., one containing authenticated token) returns the *success* set to 1 and *changes*, containing the changes proposed in this commit if the token passes all validation. In case of failure at any step, the server responds with the *success* attribute set to 0 and *message* set to a string describing the error. Exemplary responses associated with this *proto_action* can be found in Listings 5 and 6.

Listing 5. A *get_change_from_hash* response with token and challenge.

```
{
"success": 1,
"token_value": "17fe335be2244c9d01e7c4f952a170...",
"challenge": "b7725f7835b594fface23...",
"contract_address": "0xe78A0F7E598Cc8b0Bb87894...",
"abi": "\"[{\\\"inputs\\\":[{..."
}
```

Listing 6. A *get_change_from_hash* response with proposed changes.

```
{
"success": 1,
"changes": {
"boolean_data": false,
"background_color": "red"
}
}
```

- *get_config*—the server responds with the currently running configuration. This action is not critical as it depends on the system or application. As mentioned before, this action might succeed only if a specific token was authenticated. In some cases, the configuration is visible in normal operation. The returned data are not part of the configuration process, meaning that various formats can be implemented. For instance, the server may return the data encrypted with the specific public key so that only the owner of the associated private key can see the configuration.
- *get_schema*—the server returns the currently defined schema. It contains the order in which changes should be put in the JSON object, the order of fields in the commit, and the schema's hash. The client is required to construct local commits based on this response. The exemplary response is presented in Listing 7.

Listing 7. A *get_schema* response.

```
{
"changes_order": "boolean_data,background_color",
"fields_order": "changes,creator,nonce,...",
"schema": {
"additional_authorizers": "[string]",
"background_color": "string",
"boolean_data": "bool",
"creator": "string",
"depends_on: "[string]",
"nonce": "-",
},
"schema_hash": "83d6aeedddb0e...",
}
```

- *change_config*—the response follows the general rules outlined so far. It contains the *success* field indicating whether the operation was successful. In case it was not, it is set to 0, and another field, named *message*, is present containing a string describing the error. In case the operation was successful, the *success* is set to 1, and there are two other JSON objects present in the response—*commit* and *contract_details*. The *commit* object contains the *nonce* key, with the generated nonce for this particular commit, and the *hash_begins_with* key, which contains the first 12 characters of a hex representation of the commit's hash. Those 12 characters are enough for the client software to verify if the server and client agree on the commit's content. The *contract_details* contains the *addr* key with the address of the smart contract to interact with along with *abi*, which describes how to interact with it. An example of a response for this action is presented in Listing 8.

Listing 8. A *get_schema* response.

```
{
"success": 1,
"commit: {
"hash_begins_with": "ac2be6380822",
"nonce": 98430
},
"contract_details": {
"addr": "0xe78A0F7E598Cc8b0Bb87894B0F60...",
"abi: "\"[{\\\"inputs\\\":[{...",
}
}
```

- *get_abi*—the response contains the *success* field indicating whether the operation was successful and the *abi* field containing the ABI if the *success* is set to 1 and *message* otherwise.

4.4. Smart Contracts

The smart contract itself—implemented for Ethereum blockchain as a proof-of-concept—has a functionality similar to a multi-signature wallet solution with additional features. This is the part of the protocol that is responsible for authorization. The smart contract is designed in a completely service-agnostic manner. The contract itself needs to be aware of which addresses are associated with the administrators. As long as given services share the same administrators, the same smart contract can be used. The vase contract (overview visible in Listing 9) is responsible for authenticating tokens to read the proposed changes as well as actually emitting events telling the server to stage them.

Listing 9 contains the variables and function names used in the contract. However, for clarity, the actual implementation was omitted in this paper, and comments providing the high-level description were provided instead. The complete implementation used for testing and verification can be found in a public repository [7]. It is worth mentioning that the smart contracts considered in this paper are not audited and should not be considered production-ready software. They are purely a proof-of-concept implementation to verify the functionality of the proposed solution. However, they do not derive from the well-tested contracts or interfaces (such as OpenZeppelin) to limit the implementation to the logic related to the new protocol and they intentionally lack some of the features that would be required—for example, it is not pausable. It is good practice to implement a "pause" functionality in a smart contract that allows the halting of all critical operations should an unexpected event occur. For example, if there was a vulnerability identified, then it might be better to "pause" the contract until countermeasures are implemented rather than risk the potential exploitation. It should be noted that before going further with official standardization or implementing this protocol based on this proposal, the smart contracts need to be audited.

Listing 9. A base smart contract.

```
contract BaseChangeAAA {
struct Commit {
bool _needsAuthorization;
uint256 _authorizersLeft;
mapping(address => bool) _authorizers;
address _commitCreator;
bool _exists;
}

mapping(string => Commit) _hashToCommit;
```

```
address _server;
mapping(address => bool) private _isAddressAdmin;
mapping(address => bool) private
_isItADeployedAuthorizerContracts;

event BaseContractCreated(address _baseAddress);
event AuthorizerContractCreated(
address indexed _authorizerContractAddress,
string _commitHash,
address[] _authorizersNeeded
);
event ChangeCanBeStaged(
string _commitHash,
address indexed _creator
);
event TokenAuthenticated(
address indexed _address,
string _challenge
);

modifier onlyServer() {
// modifier making sure that function was invoked
// by an address (server) set as owner of this contract
}

modifier onlyAuthorizer() {
// modifier making sure that function was invoked
// by Authorizer contract
}

constructor(address[] memory admins) {
// constructor setting the owner (server) address and
// saving the addresses provided as arguments as admins
}

function authenticateToken(string memory _challenge)
public {
// emits TokenAuthenticated Event with
// the provided challenge and sender's address
}

function createCommit(
address _commitCreator,
string memory _commitHash,
address[] memory _commitAuthorizers
) public {
// Creates a commit. If~creator of the commit is one of
// the administrators then ChangeCanBeStaged event
// is emitted. Otherwise, it calls
// createAuthorizerContract function to deploy
// a new Authorizer contract
}

function createAuthorizerContract(
```

```
string memory _hash,
address[] memory _commitAuthorizers
) private {
// Deploys a new Authorizer contract and associates it
// with Commit identified by its hash
}

function someoneAuthorizedChange(
string memory _hash,
address _whoDidIt
) external onlyAuthorizer {
// Called by Authorizer contract whenever someone
// authorized a change there. This function verifies if
// it was a valid operation
}
}
```

If a user proposing the change is actually an administrator, then the base contract is all that is needed to confirm this change correctly. However, in case additional authorization is necessary, then the base contract acts as a factory for a so-called authorizer contract, which is deployed individually for every change that requires such authorization. An overview of the authorizer contract is presented in Listing 10. It might be argued that the deployment of a separate authorizer contract is not necessary, as the same functionality could be implemented in the base contract, which is a valid point. However, there are some points that need to be considered here. First of all, contract deployment is an expensive operation in comparison with function calls. The authorizer contract is deployed only if additional authorization is needed, i.e., the change was proposed by a non-administrator user. This is done in order to disincentivize any bad actor from using the contract if not actually wanting to propose a meaningful change, as such operation would be financially expensive. Furthermore, splitting the logic into separate modules (or contracts) is a common practice in software development as it helps with the readability and cognitive understanding of the code. The contract's size matters from Ethereum's point of view, as the legitimate users should be able to use it as cheaply as possible [27].

Listing 10 contains the variables and function names used in the contract. As above, for clarity, the actual implementation is omitted in this paper, and comments providing the high-level description are provided instead. In addition, the complete implementation used for the testing and verification of this type of smart contract is available in a public repository [7].

Listing 10. Authorizer contract.

```
contract Authorizer {
BaseChangeAAA _base;
mapping(address => bool) _authorizerNeeded;
string _commitHash;
bool _isOnline;

event SomeoneAuthorized(
string indexed what,
address indexed _who
);

modifier onlyBase() {
// modifier checking if Base contract invoked a function
// that uses this modifier
```

```
}

modifier isOnline() {
// modifier checking if this contract is in
// an "online" state
}

modifier validHash(string memory hash) {
// modifier checking is provided hash is equal to
// the one saved in this contract's storage
}

constructor(
address _baseAddress,
string memory _hash,
address[] memory _authorizers
) {
// Constructor creating this contract. It saves
// the commit's hash and Base contract's address.
// Additionally, it sets this contract to
// an "online" state
}

function goOffline() external onlyBase {
// Function which can be called by the Base contract
// that deployed this contract. It switches this
// contract to an "offline" state
_isOnline = false;
}

function authorize(string memory hash)
public isOnline validHash(hash) {
// Function used to authorize a change. If~commit hash
// is equal to the one expected and caller is
// an administrator then this contract calls
// someoneAuthorizedChange function in Base contract
}
}
```

The smart contracts presented in this paper as an example were implemented in Solidity programming language for an EVM-based (EVM stands for Ethereum Virtual Machine) environment. It is however possible to implement the same behavior in other programming languages for different Blockchains, as all of them are Touring-complete.

4.5. Governance

The previous description of the protocol depicts the most basic way this protocol can be implemented. It was described as such to decrease the complexity of the narrative. That description is sufficient if only one administrator is defined (which often might be the case). In this scenario, all protocol security features are void if the administrator is compromised, as the administrator has complete control over the protocol. A *Governance* mechanism should be implemented to eliminate this single point of failure. This means there should be a voting process associated with authorizing a change, i.e., only if the majority (or all) of the administrators agree on a particular change can it be implemented. The protocol allows

additional_authorizers to be defined in multiple places as the Governance requirements will be different for every application.

There might be different levels of criticality associated with a given change, and not all changes might require unanimous voting with all privileged users present. For instance, changing an internal website's background color does not necessarily require all administrators to interact with the Blockchain if that action is not considered critical. On the other hand, changing the configuration related to the product's safety parameters in a factory will most likely require all administrators (for instance, the IT department, management, and executive employees) to agree. Thus, the voting mechanism should be associated with every change to maximize security.

5. Results and Security Considerations

The protocol proposed in this paper is a service-agnostic attempt at bringing a standard-based approach to the Blockchain environment. The idea proposed by the authors is the following: the protocol operates as an abstraction layer for the configuration process itself. Although the proposed protocol shares similar functionality to the SNMP protocol [28], it differs because it is actually a functional wrapper over the configuration protocol itself. It does not dictate how the configuration should be performed, e.g., it does not require the re-implementation of such processes if they are already in use. Instead, it adds the Blockchain layer, which will add authentication, authorization, and accounting to the process. There is no pre-Blockchain protocol that would implement similar behavior, making it challenging to create a comparison. Nevertheless, it is worth stressing the critical characteristics of the protocol in a structured manner.

Moving the authentication and authorization (critical security-related operation) to the smart contract solves the most common issues that affect mainstream applications. First of all, there is no risk associated with password leakage, as there are no passwords. There is only one way to interact with the Blockchain as a user with a given address, and that is by knowing the private key associated with that address. Those keys should be known only to the account owner (like passwords). However, they are not stored by any application in any way.

Furthermore, the account address (e.g., public key) and private key are pure asymmetric encryption. Anyone can generate their own key pair. However, this does not require a user to choose his or her own private key in any way. As a consequence, it eliminates the risks commonly associated with choosing a simple password—it cannot be easily guessed. Of course, it would be possible for someone to randomly guess a private key associated with a given Ethereum address. However, it is highly unlikely, as the private key size of the key used in Ethereum is 32 bytes (Blockchains use Elliptic Curve Cryptography). The authorization aspect is covered by the Blockchain's immutability. Once a contract is deployed, its state cannot be modified by anything other than that contract's functions (if implemented). This ensures that it is always well known who is considered an administrator. In the proof-of-concept implementation, there is no functionality allowing an update of who is considered an administrator to limit the complexity. In a real-world scenario, such functionality would be needed. However, it can be easily implemented. In any case, the execution of such an update function would be public, and there would be no way for anyone to be unable to check who is a privileged user.

As all information stored on the Blockchain and every interaction with it is public, the accounting part of the security equation is covered by the Blockchain's design itself. Using one of the public Blockchains (such as Ethereum, Solana, NEAR, and others), which are widely used by people around the world, increases the redundancy. Private Blockchains, operated, for example, by a single company, are more prone to denial-of-service attacks. Furthermore, the company can be compromised by a major security incident, and it might be possible for such a Blockchain to be completely deleted. The chances of a similar situation in the public Blockchain are so slim that such a situation can be considered impossible.

However, there are challenges associated with using this protocol. The main one is that interacting with the Blockchains requires paying for the computation, which might add up to a substantial amount of money if frequent changes are required. If fees would be a significant concern, then one mitigation would be implementing and using a Blockchain with low computational fees, for example, a Solana or NEAR Blockchain instead of Ethereum, which is relatively expensive to use. It is worth stressing that, regardless of the chosen platform, fees cannot be eliminated.

The fees in Table 2 were calculated using Ganache [29], which runs a local node of the Blockchain. Those results are relatively isolated, as the costs do not fluctuate as they would in the public Blockchain. The actual costs depend on the specific Blockchain and its utilization at the given time. On the other hand, differences in cost between actions should stay relatively consistent, e.g., proposing a change costs at least 11 times more than any other action (the second most expensive operation would be authorization of a change with prior token authentication). Those results show that the protocol is relatively inexpensive for the administrators compared to the user who proposes a change. In most cases, an entity proposing a change will be a part of the same organization as the one authorizing it. However, in other cases, the external entity wishing to participate in the protocol must pay more fees.

Table 2. Blockchain interaction fees.

Action	Cost
Commit creation without additional authorization	0.00063842 ETH
Commit creation with additional authorization	0.0162945 ETH
Additional authorization	0.00091722 ETH
Token authentication	0.00055626 ETH

Another issue is the latency. Adding Blockchain as an intermediate party increases the overall security. However, it also decreases the speed of applying a change. Even if no additional authorization is required and the Blockchain immediately signals that changes can be staged, there is still a little computation required to happen on-chain, and transactions must wait to be picked up by entities creating new blocks. Only after that is completed can the server actually enforcing the change perform the necessary actions. In this case, the latency can be mitigated to some extent by choosing a Blockchain platform with relatively short block creation times and high TPS value, or in some cases, by paying more for a transaction.

Security Considerations

The most common method of authentication is a shared secret approach (e.g. passwords). However, the security of this approach inherently depends on the complexity and length of a shared secret. These features are often controlled by users. This fact introduces flaws in the system in the form of weak and fairly quick to guess passwords. Blockchain technology instead relies on asymmetric encryption, forcing users to generate their private keys, eliminating the guess factor. As a result, the only way to impersonate someone on the Blockchain is to use that person's private key. Although theoretically possible, brute-force approaches are infeasible due to the length of the private key.

The denial of service condition of the Blockchain will naturally render the whole protocol unusable as long as the Blockchain network is down. Such scenarios are possible although implausible in the mature Blockchain environments. Additionally, it is worth noting that causing a denial of Service condition on any of the critical parts of the system might also impact the protocol performance. Mainly, this is the component responsible for introducing changes and listening to the Blockchain events ("server" in Figure 1). However, even during a critical failure, when all information retained in that entity's memory is lost,

the protocol itself is not affected severely. The changes could be recreated by proposing them again.

The off-chain attack on the system is another risk worth taking into account. A system may be exploitable due to vulnerabilities residing in software not necessarily associated with the protocol itself. The underlying system may be impacted in such a scenario. However, it would not be possible to tamper with the protocol itself. The authorization logic is contained within the smart contract, which is immutable by design. After the attack, the protocol can be used in the same manner, as long as there is no leakage of the private key.

The most significant attacks on this protocol are phishing attacks. The easiest way for a malicious actor to exploit the protocol is to convince the administrator to leak his or her private key. The security provided by the Blockchain is broken when a private key is known to someone other than its owner. This risk can be mitigated to some extent by implementing a governance mechanism mentioned in Section 4.5 and utilizing multi-signature wallets. However, ultimately, it will depend on the administrator's susceptibility to such kind of attacks.

6. Discussion and Future Directions

It was already mentioned that the general idea is similar to the one proposed by Helebrandt et al. [12]. However, there are notable differences—mainly the actual environment. Helebrandt's idea was designed to operate in the private Blockchain associated with a single organization. It stores the configuration on the Blockchain itself, i.e., it is public. Furthermore, the configured devices (network equipment) had to also interact with the Blockchain, which can be considered as another layer of resiliency in the whole process. Such an operational design is absolutely fine in the private Blockchain. However, it is not scalable to public Blockchains such as Ethereum (Helebrandt's implementation uses Hyperledger, which lacks the decentralization aspect of the Blockchain [8]). In comparison, the protocol proposed in this paper was designed to be universal—it will be able to operate both on the private and public Blockchain. However, it is worth noting that the actual configuration proposal is stored in an off-chain repository, allowing to maintain a decent level of privacy even when operating on the public Blockchain. Furthermore, it can be argued that it would not be possible to retrieve the configuration from the Blockchain upon complete data loss (e.g. due to critical failure). This reasoning would be correct, however, it should be emphasized that the protocol proposed in this paper focuses on the mechanism facilitating controlled changes into configuration but not its retention. Therefore, configuration backups should be implemented regardless of this protocol.

In the previous section, the authors considered challenges regarding the proposed protocol: on-chain code execution requires paying fees, and there is inherent inertia before enforcing a change due to the latency associated with using Blockchain. Those two issues might be limited by choosing a specific Blockchain platform. However, they will never be eliminated. Based on this, the protocol might not be the best fit for selected environments where changes need to happen rapidly and frequently. On the other hand, Blockchain provides the highest possible level of security when it comes to accounting for users' actions and resilience without a single point of failure. Based on this, an ideal environment for this protocol would be mission-critical systems, which above all require consistency and transparency, for example, factories, the military sector, or the energy sector, including smart grids.

Future research on the proposed solution might focus on comparing the performance of the protocol among different Blockchains. Furthermore, there are more areas, other than configuration, which can benefit from Blockchain usage. In addition, other applications for the proposed scheme of the protocol could be proposed in order to integrate Blockchain technology with more aspects of the modern world.

Author Contributions: Conceptualization, M.B. and M.N.; methodology, M.B. and M.N.; software, M.B.; validation, M.B.; formal analysis, M.B. and M.N.; investigation, M.B. and M.N.; writing—original draft preparation, M.B.; writing—review and editing, M.B. and M.N.; visualization, M.B.; supervision, M.N.; project administration, M.N.; funding acquisition, M.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been funded by the European Union's Horizon 2020 Research and Innovation Program, under Grant Agreement no. 830943, project ECHO (European network of cybersecurity centers and competence hub for innovation and operations).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study—Blockchain Async AAA PoC Implementation—are available online in: https://gitlab.com/mrsnoug/blockchain-async-aaa-poc-implementation (accessed on 28 August 2022).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Tao, H.; Zhou, J.; Liu, S. A survey of network security situation awareness in power monitoring system. In Proceedings of the 2017 IEEE Conference on Energy Internet and Energy System Integration (EI2), Beijing, China, 26–28 November 2017; pp. 1–3. [CrossRef]
2. Borenius, S.; Gopalakrishnan, P.; Bertling Tjernberg, L.; Kantola, R. Expert-Guided Security Risk Assessment of Evolving Power Grids. *Energies* **2022**, *15*, 3237. [CrossRef]
3. Alghassab, M. Analyzing the Impact of Cybersecurity on Monitoring and Control Systems in the Energy Sector. *Energies* **2022**, *15*, 218. [CrossRef]
4. Ganguly, P.; Nasipuri, M.; Dutta, S. Challenges of the Existing Security Measures Deployed in the Smart Grid Framework. In Proceedings of the 2019 IEEE 7th International Conference on Smart Energy Grid Engineering (SEGE), Oshawa, ON, Canada, 12–14 August 2019; pp. 1–5. [CrossRef]
5. Ethereum. Ethereum Improvement Proposals. Available online: https://eips.ethereum.org/erc (accessed on 28 August 2022).
6. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 28 August 2022).
7. Bajor, M. Blockchain Async AAA PoC Implementation. 2022. Available online: https://gitlab.com/mrsnoug/blockchain-async-aaa-poc-implementation (accessed on 28 August 2022).
8. Zhang, K.; Jacobsen, H.A. Towards Dependable, Scalable, and Pervasive Distributed Ledgers with Blockchains. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2–6 July 2018; pp. 1337–1346. [CrossRef]
9. Li, W.; He, M.; Haiquan, S. An Overview of Blockchain Technology: Applications, Challenges and Future Trends. In Proceedings of the 2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC), Beijing, China, 18–20 June 2021; pp. 31–39. [CrossRef]
10. Tan, Y.; Li, W.; Yin, J.; Deng, Y. A universal decentralized authentication and authorization protocol based on Blockchain. In Proceedings of the 2020 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Chongqing, China, 29–30 October 2020; pp. 7–14. [CrossRef]
11. Ren, X.; Lin, F.; Chen, Z.; Tang, C.; Zheng, Z.; Li, M. BIA: A Blockchain-based Identity Authorization Mechanism. In Proceedings of the 2020 16th International Conference on Mobility, Sensing and Networking (MSN), Tokyo, Japan, 17–19 December 2020; pp. 98–105. [CrossRef]
12. Helebrandt, P.; Bellus, M.; Ries, M.; Kotuliak, I.; Khilenko, V. Blockchain Adoption for Monitoring and Management of Enterprise Networks. In Proceedings of the 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 1–3 November 2018; pp. 1221–1225. [CrossRef]
13. Lin, C.; He, D.; Huang, X.; Kumar, N.; Choo, K.K.R. BCPPA: A Blockchain-Based Conditional Privacy-Preserving Authentication Protocol for Vehicular Ad Hoc Networks. *IEEE Trans. Intell. Transp. Syst.* **2021**, *22*, 7408–7420. [CrossRef]
14. Perera, M.N.S.; Nakamura, T.; Hashimoto, M.; Yokoyama, H.; Cheng, C.M.; Sakurai, K. Certificate Management Scheme for VANETs Using Blockchain Structure. *Cryptography* **2022**, *6*, 20. [CrossRef]
15. Mohammad, A.; Vargas, S.; Čermák, P. Using Blockchain for Data Collection in the Automotive Industry Sector: A Literature Review. *J. Cybersecur. Priv.* **2022**, *2*, 257–275. [CrossRef]
16. Abubakar, M.; Jaroucheh, Z.; Al Dubai, A.; Buchanan, B. Blockchain-Based Authentication and Registration Mechanism for SIP-Based VoIP Systems. In Proceedings of the 2021 5th Cyber Security in Networking Conference (CSNet), Abu Dhabi, United Arab Emirates, 12–14 October 2021; pp. 63–70. [CrossRef]

17. Shahzad, K.; Aseeri, A.O.; Shah, M.A. A Blockchain-Based Authentication Solution for 6G Communication Security in Tactile Networks. *Electronics* **2022**, *11*, 1374. [CrossRef]
18. Tsoukas, V.; Gkogkidis, A.; Kampa, A.; Spathoulas, G.; Kakarountas, A. Enhancing Food Supply Chain Security through the Use of Blockchain and TinyML. *Information* **2022**, *13*, 213. [CrossRef]
19. Chiacchio, F.; D'Urso, D.; Oliveri, L.M.; Spitaleri, A.; Spampinato, C.; Giordano, D. A Non-Fungible Token Solution for the Track and Trace of Pharmaceutical Supply Chain. *Appl. Sci.* **2022**, *12*, 4019. [CrossRef]
20. Yao, Q.; Zhang, H. Improving Agricultural Product Traceability Using Blockchain. *Sensors* **2022**, *22*, 3388. [CrossRef] [PubMed]
21. Abijaude, J.; Sobreira, P.; Santiago, L.; Greve, F. Improving Data Security with Blockchain and Internet of Things in the Gourmet Cocoa Bean Fermentation Process. *Sensors* **2022**, *22*, 3029. [CrossRef] [PubMed]
22. Zhai, P.; He, J.; Zhu, N. Blockchain-Based Internet of Things Access Control Technology in Intelligent Manufacturing. *Appl. Sci.* **2022**, *12*, 3692. [CrossRef]
23. Rahman, Z.; Yi, X.; Mehedi, S.T.; Islam, R.; Kelarev, A. Blockchain Applicability for the Internet of Things: Performance and Scalability Challenges and Solutions. *Electronics* **2022**, *11*, 1416. [CrossRef]
24. Solana. Solana Explorer Page. 2022. Available online: https://explorer.solana.com/ (accessed on 28 August 2022).
25. Internet Engineering Task Force. The JavaScript Object Notation (JSON) Data Interchange Format. 2017. Available online: https://datatracker.ietf.org/doc/html/rfc8259 (accessed on 28 August 2022).
26. Group, I.N.W. Hypertext Transfer Protocol—HTTP/1.1. 1999. Available online: https://datatracker.ietf.org/doc/html/rfc2616 (accessed on 28 August 2022).
27. Ethereum. Downsizing Contracts to Fight the Contract Size Limit. 2022. Available online: https://ethereum.org/en/developers/tutorials/downsizing-contracts-to-fight-the-contract-size-limit/ (accessed on 28 August 2022).
28. IETF: Network Working Group. A Simple Network Management Protocol (SNMP). 1990. Available online: https://datatracker.ietf.org/doc/html/rfc1157 (accessed on 28 August 2022).
29. Truffle Suite. Ganache Website. https://trufflesuite.com/ganache/ (accessed on 28 August 2022).