cryptography MDPI

Article

Localized State-Change Consensus in Immense and Highly Dynamic Environments

Linir Zamir and **Mehrdad Nojoumian** *

Department of Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431, USA; lzamir2016@fau.edu
* Correspondence: mnojoumian@fau.edu

Abstract: Consensus algorithms are the building block of any decentralized network where the risk of malicious users is high. These algorithms are required to be robust, scalable, and secure in order to operate properly. Localized state-change consensus (LSC) is a consensus algorithm that is specifically designed to handle state-change consensus, where the state value of given data points can dynamically change and the new value needs to be reflected in the system. LSC utilizes a trust measurement mechanism to validate messages and also enforce cooperation among users. Consensus algorithms, and specifically LSC, can be a practical solution for the immutable and secured communication of autonomous systems with limited computational resources. Indeed, distributed autonomous systems are growing rapidly and the integrity of their communication protocols for coordination and planning is still vulnerable because several units are required to act independently and securely. Therefore, this paper proposes a new localized consensus algorithm for immense and highly dynamic environments with validations through reputation values. The proposed solution can be considered as an efficient and practical consensus solution for any paradigms with resource-constrained devices where a regular encrypted communication method can negatively affect the system performance.

Keywords: consensus algorithms; blockchain; decentralized settings; reputation; state-change

Citation: Zamir, L.; Nojoumian, M. Localized State-Change Consensus in Immense and Highly Dynamic Environments. *Cryptography* **2022**, *6*, 23. https://doi.org/10.3390/cryptography6020023

Academic Editor: Kentaroh Toyoda

Received: 13 February 2022
Accepted: 25 April 2022
Published: 6 May 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Blockchain technology began its popularity in 2008 with Satoshi Nakamoto's new peer-to-peer algorithm and his innovative way of achieving consensus among permissionless users, called proof-of-work (PoW) [1]. Ever since, more and more algorithms have emerged with similar yet different methods of achieving consensus. Famous ones are the proof-of-stake (PoS) [2], proof-of-authority (PoA), and many more. Although Satoshi's initial motivation was to use the proposed algorithm as a digital currency, it has recently been implemented in many other fields such as health care, supply chain, information sharing, etc.

Blockchain technology allows users to validate and secure immutable data that is replicated across the majority of users with a unique decentralization characteristic. Nowadays, most exchanges, whether it is a transaction or any other form of data, are monitored and accepted by a trusted third party such as banks, government agencies, etc. The reason is that, if the exchange is not monitored, some people might share false data, and as a result, make the exchange untrustworthy. The decentralized solution aims at solving this issue by executing a consensus algorithm designed to validate any report of data without the need for a trusted third party. More specifically, information is stored on immutable blocks containing chains of data. Every given time, a new block of information is added to the chain by an approval process, called *mining*. Various consensus algorithms have different mining processes; however, the end goal is the same for all of these schemes. In some of these consensus algorithms, the validation of data is predefined in a way that users can

execute it automatically by running a simple server that can operate independently. This is the primary reason that new blockchain applications have been recently developed in emerging fields such as robotic swarms and autonomous driving [3].

1.1. Consensus Algorithms

The blockchain data structure was first introduced in 1990 by Stuart Habert and W. Scott Strornetta [4]. The primary goal was to timestamp digital documents, making them tamper proof. Over the years, blockchain data structure has expanded to many other fields such as economy, e-voting, assembly line, etc. Many of these applications do not rely on cryptocurrency exchange, rather, the users exchange information in a decentralized fashion. Different decentralized applications are classified as either public or private. Public systems do not have any restriction on peers and they do not require any authentication process for joining the network or initiating trades. The public decentralized systems are maintained only by the public community, which means a higher level of decentralization. Private or permissioned systems operate under the leadership of a group, often called *consortium*, which is the only group that can manage the system. They implement a registration process that a user has to go through to be able to execute transactions.

Blockchain technology offers a decentralized environment without the need for a third-party authority to authenticate data. To successfully operate, a *consensus mechanism* is required. Over several years, many consensus algorithms were introduced in the literature, each of which with its own properties. The primary goal in all of these algorithms is to have a fast and reliable mechanism to authenticate data in the form of transactions.

As an example, we can refer to the popular PoW consensus mechanism that was proposed in [5]. This mechanism is used by Bitcoin [1], which is the most prominent blockchain system today. In this platform, whenever a new block is mined, the first miner who completes the mining process becomes the single authority to add the new block to the chain. The mining process includes finding a proof in the form of a hash value for the block. There exists a difficulty factor associated with the hash calculation. The first miner who finds this proof can broadcast it to the rest of the network, where validators can then validate it and decide whether it is a valid proof or not. Proof-of-stake [2] is another example of a consensus mechanism that handles transaction validations with some improvements over the classic PoW. In this mechanism, an auction is carried out and the winner becomes the miner, i.e., a leader is selected based on his/her bid or how much money he/she is willing to put at *stake* in order to win the auction. Unlike PoW, in PoS, whenever a new block is mined, new coins are not distributed in the system. However, miners are rewarded with a transaction fee. Proof-of-reputation (PoR) [6] is a reputation-based consensus algorithm where each user is given a reputation value that is based on its activity in the network. This consensus algorithm is similar to the reputation-based mining paradigm proposed earlier in [7]. In PoR, block miners are elected based on their reputation values and each block is then validated via reputation-based voting. The idea of assigning reputation values to agents can lead to faster consensus as well as data validation.

The algorithms presented above laid the foundation for the *information consensus* algorithms, which are mostly used in cooperative control of multiagent systems. The basic idea is that agents update their local information states based on the state of their local neighbors in a way that the final information state of all units is the same. Information consensus has many applications in autonomous systems, smart vehicle communications and robotic swarms, where the system is required to achieve a consensus among all of its units, also known as agents.

1.2. Information Sharing

In information sharing frameworks, the state of a given agent is shared among multiple agents for decision making. Proposed paradigms in [8–12] are examples of multiagent systems with information sharing algorithms in which a shared data consensus is achieved. In many of these algorithms, the consensus is set as a constant value that may not be ap-

plicable to a dynamic environment, where information is time-bound and can be changed every so often. A dynamic state-change environment was presented in [11], where the shared information is a flying configuration between agents in the system. This implementation, however, does not handle adversarial agents that can completely change the dynamic of the system. In addition, most of these algorithms require all agents to achieve the same single-state consensus, while in real-life scenarios, it is possible that only a few agents have access to a time-sensitive reference state. In addition, communication among agents might be unreliable, and therefore, partitioning is required in dynamic environments. Ref. [9] presented an information consensus framework for multiple vehicle systems, where a global consensus is required. Jadbabaie et al. [8] proposed an innovative solution for communication among agents with neighbor changes, which allows for a dynamic state-change environment and is suitable to handle partitioning. However, this design achieves a consensus over all agents for a single environmental data and cannot be broken down into separate data points. While most papers in the literature utilize a set of n agents and directed graphs as a model of interaction among these agents, our paper utilizes an n-tree as a communication model that allows a faster consensus to be achieved and requires fewer interactions among agents, i.e., making it potentially more efficient.

1.3. Our Motivation and Contribution

This paper provides an innovative information sharing consensus algorithm, called *localized state-change* (LSC), that is designed to deal with state-changes in highly dynamic environments through local consensus where any state-change is only visible to a subset of agents. This is a more realistic model for many applications such as cooperative driving among self-driving cars, coordination and planning among drones, etc. For instance, self-driving cars can share the state of roads in a cooperative driving context; however, in a large metropolitan area, it is not possible to achieve a global consensus since any local state-change is only visible to a subset of self-driving cars. We achieve the local consensus by utilizing agents' reputation values and a voting (or signature) threshold that is defined based on the reputation value of the agent observing the state-change in a specific area as well as the total number of agents in the system.

This new design is a localized voting consensus algorithm that can operate in the presence of adversarial agents similar to the modified version of the Raft consensus algorithm, called ISRaft [13]. The proposed new scheme utilizes a leader–follower framework and incorporates data validation by assigning reputation values to agents. It utilizes these properties to achieve data validation on the state-changes in highly dynamic environments. The new design is intended to create potentially infinite scalability and process thousands of state-change transactions per second, even with a large number of agents in the network. The design can also be implemented in resource-constrained devices, making it ideal for emerging autonomous systems or any applications where agents might have a limited computational power, e.g., small drones and robots, among others.

Most crypto applications nowadays, including many state of the art consensus algorithms, are designed to be implemented in a cryptocurrency environment where a consensus over a shared transactions ledger is required. There are, however, many other possible applications that have only recently emerged such as supply chain, communication and decision making, etc. In these applications, the shared ledger contains the required data and the consensus provides the immutability property. LSC was designed with the ability to verify as well as validate data shared across multiple agents, i.e., validation on the content of the transaction, not just on the transaction execution. Unlike other existing state-of-the-art consensus algorithms, LSC provides a combination of the following features:

1. **Voting Threshold:** LSC uses reputation for different purposes. That is, agents gain reputation by acting honestly and lose it otherwise. This property is similar to many state-of-the-art consensus algorithms. Additionally, reputation is used to define the required number of agents, i.e., voting threshold needed to achieve a localized

consensus. As shown in Equation (2), a higher reputation value lowers the required number of votes.

2. **Local Consensus:** LSC is designed to handle quick changes in immense and highly dynamic environments. This means that changes can happen quickly and simultaneously. Instead of achieving at least 51% consensus, LSC uses a localized consensus algorithm to verify and validate changes within the environment. Once a local consensus has been achieved, it can be used to achieve a 51% consensus, very much like any other consensus protocol.
3. **Data Validation:** LSC helps agents validate the content of the shared data. It utilizes both localized consensus and reputation values to guarantee any changes in the environment are reflected on the chain. As stated earlier, validation is related to the content of the transaction, e.g., a road is blocked, not just on the transaction execution.

To the best of our knowledge, there is no other consensus algorithm with these properties. Table 1 provides a comparison between state of the art consensus algorithms and LSC.

Table 1. Summary comparison of blockchain consensus algorithms. Partially taken from [14].

Consensus Algorithm	Designing Goal	Type	Verification	Local Consensus	Data Validation *
PoW [1]	Sybil-Proof	Permission-less	Work or Hash	No	No
PoS [2]	Energy Efficient	Permission-less	Stake	No	No
PoA [15]	Combination of PoS and PoW	Permissioned	Vote and Work	No	No
PoR [6]	Reputation-Based Consensus	Permissioned	Vote	No	No
pBFT [16]	BFT-Based Consensus	Permissioned	Vote	No	No
Raft [17]	Accessible Paxos	Permissioned	Vote	No	No
ISRaft [13]	Raft for Malicious Environments	Permissioned	Vote	No	No
LSC [This Paper]	Localized State-Change Consensus	Permissioned	Vote or Localized Vote	Yes	Yes

* Data validation on the content of the transaction, not on the transaction execution.

The remainder of this paper is organized as follows. Section 2 presents a summary of the LSC consensus algorithm with its state-change method, Section 3 covers the consensus algorithm in detail, Section 4 provides a detailed analysis and finally, Section 5 summarizes the concluding remarks.

2. Preliminaries, Notations and Definitions

LSC is a *scalable decision-making election-based* consensus algorithm. This means, unlike many other blockchain implementations that focus on achieving a consensus for the transactions on the ledger, LSC is designed to achieve consensus on the validity of the data that is written on the chain. In other words, LSC is designed to provide decision-based consensus by validating the information sent among users.

This paper nonetheless describes LSC under the context of a *permissioned* blockchain. Users must have prior knowledge of all other users and their signatures. When a new user joins the network, it goes through a series of enrollment processes where it is assigned a pair of cryptographic keys and a reputation value. Among the protocol's immutable communication, LSC also includes a state-change validation process where a group of users can change a state of some external information. It then goes through a validation process that includes the comparison of reputation values and state reading. If the validation data is accepted by the majority of users, the state changes and it is reflected in the blockchain. Two major properties that are required for achieving the decision-based consensus are

authentication and *reputation*. We assume a resilient reputation model [18] is utilized in our scheme for the trust management. For the sake of clarity, Table 2 defines our notations.

1. **Authentication:** Users are assigned a pair of cryptographic keys used for authentication of messages. When a user joins the network, it shares its public-key with all other users. When the user sends any message, it then signs it using its private-key in an asymmetric encryption fashion.
   Formally, for key-generator G and security parameter k, a public-key (pk) and a corresponding private-key (sk) are generated by:
   $$(pk, sk) \leftarrow G(1^k)$$
   Let S be a *signing* function that takes a private-key (sk) and a string (x) and returns a tag value (q).
   Let V be a *verifying* function that takes a public-key (pk), a string (x) and a tag value (q) and returns *accepted* if the tag value matches the expected value, or returns *rejected* otherwise. Users verify messages (x, q) and reject any message that does not include a valid signature. For the purpose of this model, any key-pair encryption algorithm, e.g., RSA, can be used.
2. **Trust:** T_i^j is the trust value assigned by $\mathtt{u_j}$ to $\mathtt{u_i}$, which is a numeric value that quantifies how trustworthy $\mathtt{u_i}$ is. Note that trust is a personal quantity for a player from the perspective of another player, whereas reputation is a social quantity for a player from the perspective of a set of players [19]. This value is defined by:

   Definition 1 (Reputation Value)**.** *Let T_i^j be the trust value assigned by $\mathtt{u_j}$ to $\mathtt{u_i}$. Let T_i be the reputation function that illustrates how trustworthy $\mathtt{u_i}$ is from the perspective of all users [20]:*
   $$T_i = \frac{1}{n-1} \sum_{j \in \{1,\cdots,n\}, j \neq i}^{n} T_i^j \tag{1}$$

For the sake of simplicity in this article, we just use the reputation value as a public parameter. Agents then update a reputation value when an honest or a dishonest message is broadcasted. In our setting, the reputation value is used for data validation and state-change acceptance.

These two properties are what makes LSC functional in the presence of adversarial users. They help the network in achieving consensus for any state-change task. In the literature, it is shown that reputation can be utilized as an effective model parameter to prevent adversarial activities [21] or incentivize blockchain miners to avoid dishonest strategies [22,23].

As stated earlier, the LSC protocol is designed to handle state-change consensus for an environment with multiple parameters, each with its own state and data. Users on the network are tasked with achieving consensus over different states. Assuming states can be changed by the environment, it is important for the LSC protocol to recognize any state-change and update it on the blockchain. A proper example of such an environment is a road system where each road can be in either of three states: open, semiopen, or closed. The state of any road can change depending on the amount of traffic it holds. Users can be vehicles driving and reading the environmental data. When a vehicle recognizes a different state on a road, it can initiate the state-change consensus to update the new state on the blockchain. Vehicles that contribute and verify the state change are rewarded by increasing their reputation values.

Table 2. Notation Used.

Notation	Description
$\mathtt{u_i}$	ith agent
T_i^j	Trust value assigned by j to i
T_i	Reputation value of i
$\mathtt{ET_i}$	ith Election timeout
$\mathtt{PK_{u_i}}$	Public-key of user i
$\mathtt{Sig_i}$	Signature of i
$\mathtt{d_j^i}$	State of data point j as seen by user i
$\mathtt{M[Sig_i, d_j^i, t_1]}$	Message M signed by i with state s
L	Maximum number of nearest agents that can validate a message
h_i	Voting/signature threshold for approval of messages sent by agent i

3. Our Proposed Localized State-Change Consensus Mechanism

In this section, we cover the LSC consensus protocol as well as the five main communication algorithms. The main goal of LSC is to allow immutable and real-time decision making between agents on the LSC network. The second goal is to validate messages with adjustable reputation values that determine the validity of the agent. LSC achieves consensus over a set of messages through a Byzantine fault-tolerant (BFT) protocol. This protocol supports message authentication, partitioning and fast computation, which makes it ideal for resource-constrained devices.

3.1. Design Overview

LSC consensus protocol is a method where all agents hold an agreed upon ledger containing data. This ledger is constructed of blocks forming a chain. An agent can be in either of the three main roles: (1) follower, (2) candidate, or (3) leader. It is also possible to be an active validator, however, it is a temporary subrole. A leader is the only authority that can concatenate new blocks to the chain, acting as the *miner* of the newly concatenated block. Followers and candidates can add new information to a new block by sending the data to the leader.

LSC aims at achieving consensus among agents in a highly dynamic environment with a set of n agents where $\mathtt{U} = \{\mathtt{u_1}, \mathtt{u_2}, \ldots, \mathtt{u_n}\}$. Communications among agents are accomplished using five main algorithms. For the sake of simplicity, we assume that the communication is conducted on a secured channel without the possibility of having a man-in-the-middle attack. This, however, can be easily addressed in future works by using verifiable protocols. The five algorithms are:

- *RequestVote:* Initiated by a candidate agent and it is sent to all other agents. This message is sent as part of the *election process*.
- *StateChange:* Initiated by any agent and it is sent to L-nearest agents that can validate a given message. This message consists of the data point ID, new state, timestamp and signatures of agents approving the data. When a total of h_i agents sign the message sent by agent i, it is sent to the leader to be added to the next block.
- *AppendBlock:* Initiated by the leader at the end of every *leadership term*. This message contains the new block to be added to the chain. The agent receiving this message is required to respond with a signed approval message.
- *CommitBlock:* Initiated by the leader after receiving *AppendBlock* approval from the majority of agents.
- *Heartbeat:* Initiated by the leader and it is sent to all other agents. This message includes signed votes from the election, latest block number and leadership-term timer counting down to the end of the term. This message is sent periodically.

The number of required validators h_i for a *StateChange* message is defined as:

Definition 2. *Suppose n is the total number of agents in the network and $T_i \in [0,1)$ denotes the reputation value of agent i. The number of required validators for the local StateChange message sent by agent i is calculated by:*

$$h_i = \lfloor \frac{n \times (1 - T_i)}{4} \rfloor \quad (2)$$

A message that is proved to be correct rewards the agent by increasing the agent's reputation value. These five algorithms make our proposed protocol easy to implement in almost any resource-restraint device. Figure 1 presents a flowchart of the leadership term process.

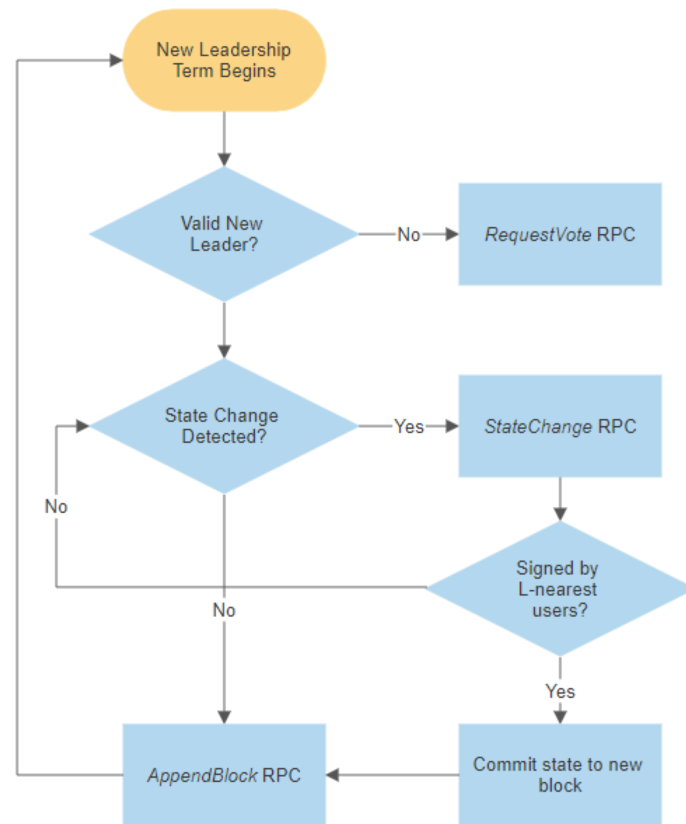

Figure 1. Leadership term flowchart.

3.2. Participants

There are four roles in the LSC network: follower, candidate, validator and leader.

1. **Follower:** A follower is the basic and most common role in the LSC network. The followers' role is given automatically for any agent who joins the network after the registration process. Followers are what drives the system to a consensus by taking part in the election process as voters, as well as by verifying any state-change event.
2. **Candidate:** Candidates' role is given to followers who initiated the *election process* after no *heartbeat* message has arrived over a fixed period of time, called *election timeout*, or when the *leadership term* has ended.
    - *Election Timeout:* Randomized local variable for any agent who joins the network.
    - *Leadership Term:* Fixed global variable

    Candidates compete to become the next leader, who is then rewarded with a significant increase in reputation value. Agents assign the candidate roles to themselves willingly when the conditions are met.

3. **Leader:** A leader is the highest authority who is responsible for sealing the next block on the LSC network. A leader is elected during the *election* process, and there can only be one leader for every newly generated block. Since the leader cannot reasonably be expected to maintain a fully synchronized communication with all other agents, it is expected that the followers are able to rebroadcast leader messages, excluding the heartbeat.
4. **Validator:** A validator's main purpose is to verify that a given data point's state has changed and that the *StateChange* message is true and valid. The process is fairly simple. Validators add their signature to the original *StateChange* message and then share it with nearby agents, who then also obtain the validator role.
   This is a dynamic role, and any agent can become a validator as long as it is close enough to the data point and to another validator agent.

3.3. Election Process

The first step of the protocol is to elect a leader in a *leader election* procedure. A leader is required to send periodic *heartbeat* messages to all other agents in the network. This message contains the signed signatures from the latest election, latest block number and leadership-term timer. This message acts as a leadership proof. Agents receiving this message validate the votes and check if the latest block number is at least equal to the block number on their database. When the timer reaches 0 or when no heartbeat messages have arrived over the expected period, a new election process begins.

At the beginning of an election, a follower agent changes its state to a candidate, and it begins sending out signed *RequestVote* messages, including the latest block number to all the agents it can contact. The latest block represents the term number of the chosen leader. For every new term, a new leader is elected. Agent $\mathtt{u_m}$, who receives a *RequestVote* message, initially checks the following conditions:

- The agent did not receive any *heartbeat* messages from the current leader.
- The candidate agent is not the leader of the current term.
- The block number from the *RequestVote* message is at least equal to the agent's latest block plus 1.
- The message has a valid signature.

If all the conditions are met, the voting agent holds its vote for a fixed period of time equal to *election timeout* $\mathtt{e}$. During this time, if any other *RequestVote* messages arrive, it once again checks if the conditions are met. If so, it then compares the reputation value of the candidate agents. At the end of $\mathtt{e}$, the vote is sent only to the agent with the higher reputation value. Algorithm 1 illustrates the voting response process for an agent after receiving a *RequestVote* message.

The process ends when a candidate receives the majority of the votes, that is, at least $n/2 + 1$ votes for a network of n agents. At that point, the elected leader starts sending out *heartbeat* messages to the network, thus completing the election process.

Algorithm 1 RequestVote Response.

Require: *RequestVote* Message (RV_i) and Agent (u_i)
Ensure: Accept or Reject RV

```
1:  t ← 0
2:  RV ← RV_i
3:  Decrease_reputation(T_i^m)
4:  while t ≠ e do                                  ▷ Election Timeout
5:      if RV.block_num <= len(chain) then
6:          Reject RV
7:      else if V(u_i.pk, RV, S(RV)) == reject then
8:          Reject RV
9:      end if
10:     if new RV_j from agent u_j then
11:         Decrease_reputation(T_j^m)
12:         if T_i^m > T_j^m then                   ▷ Compare reputation
13:             Reject RV_j
14:         else
15:             RV ← RV_j
16:         end if
17:     end if
18:     t ← t + 1
19: end while
20: Accept RV
```

3.4. Data Point's State Architecture—Local Consensus

LSC runs on an environment with a set of m data points $S = \{d_1, d_2, \ldots, d_m\}$. These data points are given per environment and can be changed when redeploying the system in different environments. A single data point can be in any number of states. For the sake of simplicity, we define these states as numerical values; however, it can be any data type. As in real life environments, a data point's state can change independently and randomly by conditions that the system is not necessarily aware of. It is up to the system to recognize the change and validate the data on the blockchain. The process goes as follows:

1. An agent u_k recognizes a change in the state of some data points d_j in the environment.
2. u_k then sends a *StateChange* message to the L-nearest agents, called **validators**. This message contains the data point ID, new state and a timestamp.
3. Any other agent receiving the *StateChange* message can decide whether to add its signature or not. When an agent adds his signature, it then sends the newly signed *StateChange* message back to the sender and to its nearest L agents.
4. Steps 2 and 3 are repeated until a total of h_i agents sign the original *StateChange* message. The value of h is calculated by the reputation values of the agents and by the total number of agents (Definition 2).
5. The h_i-th agent sends the message to the elected leader to add it to the next block.
6. Once a new block is mined, all other agents in the system update the state of data point d_j to the new state.

This process happens every time an agent recognizes a data point's state that is different from the state written on the blockchain. Agents who validate the state change and add a signature to the original *StateChange* message are called *validators* and are rewarded with reputation value upon a successful state change. The number of validators for each state change is given at the design level, and can be changed based on the reputation value of the sender, whenever the system redeploys, or by forking. The reason for this localized consensus is to prevent overloading the leaders with any state change and to prevent a possible distributed denial of service (DDoS) attack. Algorithm 2 covers the process of *StateChange* message signature. This process can also be referred to as the localized consensus process.

Agents near any state change are also likely to sign and send many *StateChange* messages from different agents who repeatedly send the signed message. If the message has already been signed by the recipient, it neither signs, nor sends it again. Another possible scenario is to have multiple h_i-length signed messages of the same origin that are sent to the leader. In this case, the leader only adds the first message and rejects any message with the same origin. This process can be represented by an h_i tree structure in which each node is an agent who signed the message and sent it to L other nodes. Figure 2 illustrates an example related to this matter.

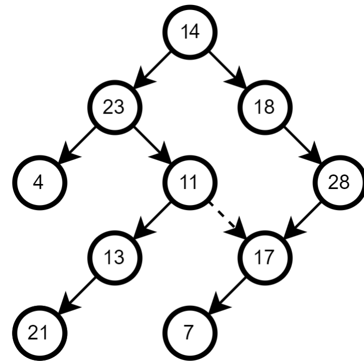

Figure 2. Representation of L tree structure.

Here, $\texttt{L} = 2$, meaning that at most 2 neighboring agents can validate a message, and a total of $h_{14} = \lfloor \frac{30 \times (1-0.333)}{4} \rfloor = 5$ signatures are required to validate any messages sent by agent 14, where $n = 30$ and $T_{14} = 0.333$. The numbers within each node represent an agent ID. We can see that there are 3 valid tree paths: $14 \rightarrow 23 \rightarrow 11 \rightarrow 13 \rightarrow 21$, $14 \rightarrow 18 \rightarrow 28 \rightarrow 17 \rightarrow 7$ and $14 \rightarrow 23 \rightarrow 11 \rightarrow 17 \rightarrow 7$. In this example, agent number 17 signed 2 different chains. This is valid since it is the first signature on either path. The first path to arrive at the leader is likely to be the one added on the next block.

Algorithm 2 StateChange Signature.

Require: *StateChange* Message M
Ensure: Signed message or Reject
1: **if** $if V(pl, M, S) = rejected$ **then** ▷ Sig validation
2: Exit
3: **end if**
4: $t \leftarrow 0$
5: **while** $t! = mt$ **do** ▷ Message timeout
6: **if** $d_i == M[d_i]$ **then return** $M[\text{Sig}, d_i, t]$
7: **end if**
8: read d_i ▷ Keep reading while message not timed out
9: $t++$
10: **end while**

3.5. Block Generation/Mining

Once a leader has been elected for some term t, it begins the preparations of adding a new block to the chain. At the end of every *leadership term*, a new block is required to be appended to the chain in a process called block generation or mining. Every block on the chain has the following data:

- **Block Number:** Additionally represents the term number of the current block.
- **Block Leader:** The agent who mined the block.
- **Timestamp:** The time when the block was mined.
- **State Changes:** Any data point that was changed is added to the block. A list of state changes with the original message and signatures is added.

- **Previous Block Hash:** The hash value of the previous block.

The blocks are hashed and connected by the appropriate block number and the hashing value of the previous block. There can only be one leader for any single block. Followers can add data to the block by sending a signed *StateChange* message consisting of a list of the new data point's states. Each of these items holds the data point ID, new state value of the data point, original author of the message and the validators' signatures. The number of signatures per message is defined by h_i and can be different among agents based on their reputation values. Agents with higher reputation values have to provide more signatures, hence, more validators are required to verify the authenticity of the message. A leader adds the state-change message to the block if and only if:

- Enough validators have included their signatures.
- The originator of the message has not already been included in the block.

There can be a case where the leader does not obtain any *statechange* messages throughout its *leadership term*. In this case, the leader simply mines an *empty* block that does not contain any state-change data. Figure 3 shows an example for block properties in LSC.

Once a block is ready to be appended at the end of the *leadership term*, the leader sends the *AppendBlock* to all agents in the network. This message includes the next new block as well as the signed votes from the election in which it won. This prevents anyone from trying to disguise themselves as the leader. When an agent receives the *AppendBlock* message, it checks the following:

- Votes are legitimate.
- Hash value of the previous block and the block number fit the latest block in the chain. If not, the agent updates the chain from nearby agents.

If the conditions are met, the agent sends a signed *AppendBlock* approval back to the leader. If and when the majority of agents in the network send the approval, the leader can then send the *CommitBlock* message including the signed approvals. Only upon receiving these messages can agents commit the new block to the chain.

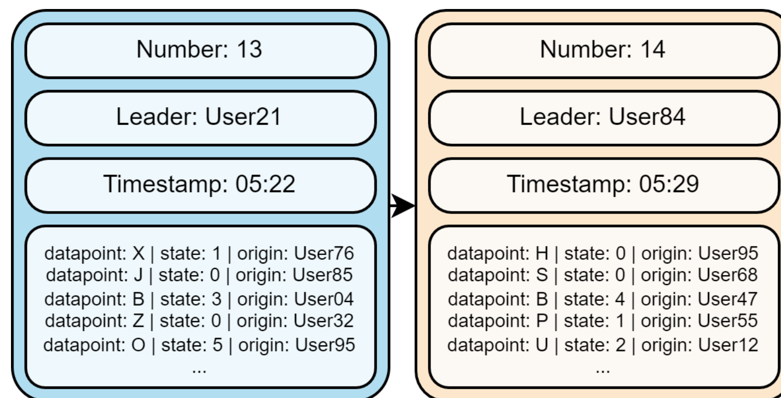

Figure 3. Blocks example in the LSC infrastructure.

3.6. Trust and Reputation

Reputation plays a significant role in verifying messages, and it can be calculated for a single agent by averaging the trust values of all other agents. Reputation value can be any decimal number in the interval $[0, 1]$ (in our implementation, we set it to have a total of 4 decimals), and it represents the confidence in which agents rely on each other, i.e., 0 being nontrusted and 1 being highly trusted. These values are unique and do not have to be symmetrical, i.e., for agents $\mathtt{u_i}$ and $\mathtt{u_j}$, $\mathtt{T_i^j} \neq \mathtt{T_j^i}$ [24].

When a follower sends a message to any other agent in the LSC network, the recipient immediately decreases the reputation value of the sender by a small amount, called **reputation cost**. However, once the message has been verified and added to the blockchain, the sender's reputation value increases by a margin larger than the cost. This is to prevent

agents from sending unauthorized messages and to prevent possible DDoS attacks on the network. Agents constantly change the reputation values of other agents over the network based on interactions in the network. Table 3 covers the reputation change for different messages in detail.

Table 3. LSC messages and their reputation costs.

Message	Reputation Cost	Details
RequestVote	✗	Candidates send this message during the election process. This does not cost any reputation since we want all agents to have a chance of winning without the risk of constantly losing on reputation values.
StateChange	✓	Initiated and sent to L-nearest agents, called **validators**. When a validator does not approve the state change, either by observing or by insufficient reputation values, it decreases the reputation value of the origin agent as well as agents who signed it. However, if the recipient approves the message and **signs** it, it increases the reputation value.
AppendBlock	✓	This message is initiated by the leader and is part of the mining block. This message includes the signed votes from the latest election. Agents decrease the reputation value of the sender in the case if the signed votes are **not** valid.
CommitBlock	✓	This message can only be initiated by the leader after the *AppendBlock* message was successful. Agents decrease the reputation value of the sender in the case the included approvals are **not** valid.
Hearbeat	✗	This periodic message does not affect the reputation value.

Once a new block is committed on the blockchain, the reputation values of all agents who originated and signed a *StateChange* message on the new block increase as a reward in a process, called **block reputation increment**, where all agents update the state of the changed data point on the new block.

4. Technical Analysis or Our Proposed Solution

We now evaluate the effectiveness and scalability of LSC followed by security analyses.

4.1. Effectiveness

The effectiveness of the protocol can be measured by the total amount of messages that are sent among different agents on the network. LSC proposes a simple five-message protocol to limit the total required bandwidth and memory per agent. The most commonly used message over a group of agents is the *StateChange* message that requires the signature of h_i validators. With the total number of validators and the total number of nearest agents L, we can easily calculate the maximum number of messages required for a single validation process for agent i.

$$\sum_{k=1}^{h_i} L^k = \frac{L - L^{h_i+1}}{1 - L} \tag{3}$$

For example, with $L = 2$ and $h_i = 5$, the maximum number of messages to be sent during this validation process is 62. To calculate the time complexity of the total required *StateChange* messages, we start by setting h_i to be the expansion value from Definition 2. Since we know that $L > 1$, we can calculate the time complexity to be $O(L^{h_i})$, which is the expected complexity from the *geometric series*, and it depends on the height of the tree, noted as h_i.

4.2. Scalability—A Storage Analysis

The generic assumption of the blockchain is that each agent can read a committed block and update the data point with the new state data. When a new block is committed, it is under the assumption of the LSC protocol that each agent stores the new block along with all previous blocks back to genesis. This assumption, however, can have some issues, especially when working with systems that can have limited memory capacity. This issue creates a scalability issue if the system is required to operate for a long period of time with many agents.

The memory requirement of each agent depends on the state-change content, the number of state changes per block and how frequently a new block is mined. More agents means more validators for any state change. As Definition 2 shows, an increase of `n` increases the value of h_i.

Let us denote the size of the state change message as $\mathtt{St_w}$, number of state changes per block as `St` and the weight of the block header as H_w. A single block weight B_w can be calculated as follows [25]:

$$B_w = \sum_{j=0}^{\mathrm{St}} \mathrm{St}_w^j + H_w \tag{4}$$

Next, from experimental evaluation, we know that the weight of a single *StateChange* message ($\mathtt{St_w}$) is ≈ 1 KB, header (H_w) is ≈ 2 KB and an average of 200 *StateChange* messages can be added per block. This value is averaged by the amount of data a single agent can receive and handle. From (2), we can then calculate the total memory requirement for a single block to be ≈200 KB. Next, let us denote the length of the leadership term as `Lt`. This value depicts how fast a new block is added to the chain. Different `Lt` values can drastically increase or decrease the weight of the LSC chain. Figure 4 shows the exponential growth in memory size when increasing `Lt`. Agents are then required to hold more than 2 GB of chain data per day on average, which can cause a problem for resource-constrained devices that are required to operate for a long period of time. This problem, however, can be addressed in either two ways:

1. Store the blockchain data on a dedicated cloud server. Agents can operate by utilizing their private-keys for reading and mining. The cloud data can be hashed and stored for chain validation processes.
2. Set a group of dedicated full-node agents who hold the full chain, unlike regular agents who hold only a snapshot of the chain.

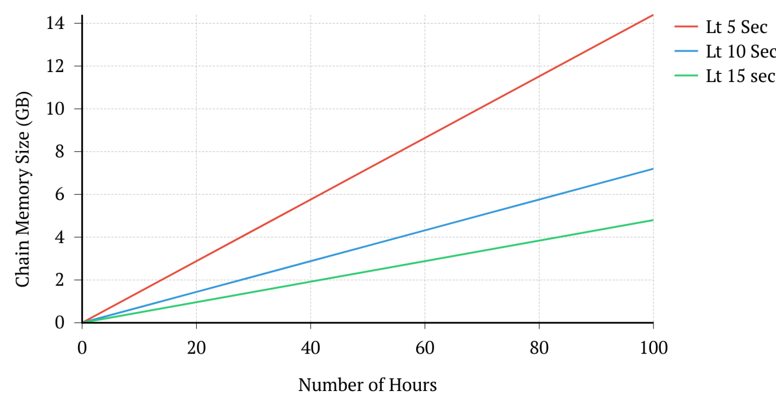

Figure 4. Required memory for chains with different `Lt` values.

4.3. Implementation

To validate our proposed algorithm, we implemented it on several AWS instances, EC2 t3.small, with 2 vCPUs and 2 GB of memory. A total of 20 instances were deployed as we found this to be an appropriate number based on our design and analysis. Each instance was given a unique ID and a set of random operations to simulate data reading and environmental changes. A small number of these instances were given an *adversarial* tag, i.e., the set of operations included some adversarial ones, such as message drop and false information sending.

4.4. Security Analysis

For this protocol to function, we have to assume there are enough honest agents on the network. This number depends on the total number of agents as well as the average number of validators per message.

False Validation Attack. Validators are essential for the protocol to function properly. The number of validators per *StateChange* message h is determined by the reputation value and the total number of agents. This value is usually small since it only requires a local consensus. This means that a small number of agents can gather and send a false state-change message only to have their reputation values to be increased in the block reputation increment process. However, it can be easily detected as the data is written on the chain and can be accessed by anyone when a malicious activity is detected. Once recognized, it is then easy to isolate and disregard any previous data made by that group. An immediate solution to this attack can be to assign a special trusted validator to a given set of data points. This validator's signature is required for any *StateChange* message that is sent regarding any data point in the designated set. It can easily be implemented in a trusted blockchain environment.

Partitioning. Partitioning can happen when a group of agents is separated from the main group by natural or malicious means. The partitioned group can miss a new leader election, new blocks committed and state changes. This problem, however, can be addressed as follows. A partitioned group of agents will not receive any *heartbeat* messages, causing them to start a separated leader election process followed by a separated state change and block mining. Once the partition is removed, an agent in the partitioned group might see two different chains—one from the main group and the other one from the partitioned group. An agent always follows the chain with the higher block number. Therefore, any state changes that were added to the shorter blockchain are removed. Any data point's state that was removed is likely to be added at a later point, as long as the state is different from the latest state on the chain.

Block Withholding Attack. Block withholding can occur when the current leader does not publish the latest block in time. Once a new block is mined, participants can update the data point's state to reflect the state on the blockchain. Validators are also rewarded with increased reputation values. When a leader holds the block, it is taking a risk of having its reputation value reduced significantly, making it unlikely for him to be elected as the leader again. Block withholding within the network can be easily detected by any agent who does not receive an *AppendBlock* message within the leadership-term period. The effect of not publishing a block in time can be a momentarily delay in the state update, however, the state is updated on the next block with a different leader.

Impersonation Attack. Impersonating an agent on the network can have a major negative effect. It can change the leader's votes, invalidate data and have many more consequences. However, this attack is not possible in LSC because of the nature of the permissioned network and the registration process. Any communication is required to be signed with the sender's private-key pk. This prevents any malicious impersonation activities. On the other hand, if an agent acquires a hold of a different agent's private-key, it can easily impersonate that other agent. This problem is true for any other system that relies on private- and public-key encryption schemes.

Distributed Denial of Service Attack. Distributed denial of service attacks are a very common attack on systems that rely on communications. In general, this attack is an adversarial attempt to disrupt the normal activity of other agents by sending frequent messages to overwhelm a node and cause a traffic jam in the communication. In LSC, any message is priced with some reputation values. This means that a malicious agent's reputation value decreases as long as it continues sending messages. Once a reputation value reaches the lower threshold for an agent, any other messages sent by it is immediately dropped, i.e., stopping it from overwhelming the network.

5. Concluding Remarks

This paper provides a new information consensus algorithm for immense and highly dynamic environments using localized consensus with validations through reputation values. This algorithm can also be implemented in resource-constrained devices, such as autonomous systems or other smart devices that communicate over a network without the need for human intervention. Validation and authenticity of messages is achieved by first initiating a local consensus using cryptographic communication means and the usage of reputation values. The consensus can then be expanded globally when the state-change has been verified. LSC assures confidentiality, integrity and validity of messages on the blockchain among all agents.

In our future work, we will test the implementation of LSC in a real-world environment using communications that can be vulnerable to different side-channel attacks. We will validate the hardware-layer security of the system and measure its efficiency in an adversarial environment. We will also explore the possibility of adding more roles to the system to increase its efficiency and speed when implemented as a decentralized autonomous organizations (DAO).

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

Funding: Research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-18-1-0483. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

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

References

1. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* **2008**, 21260.
2. King, S.; Nadal, S. Ppcoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. Self-Published Paper. 2012. Volume 19. Available online: https://bitcoin.peryaudo.org/vendor/peercoin-paper.pdf (accessed on 7 February 2022).
3. Yuan, Y.; Wang, F.Y. Towards blockchain-based intelligent transportation systems. In Proceedings of the 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, Brazil, 1–4 November 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 2663–2668.
4. Haber, S.; Stornetta, W.S. How to time-stamp a digital document. In Proceedings of the Conference on the Theory and Application of Cryptography, Aarhus, Denmark, 21–24 May 1990; pp. 437–455.
5. Dwork, C.; Naor, M. Pricing via processing or combatting junk mail. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 16–20 August 1992; pp. 139–147.

6. Gai, F.; Wang, B.; Deng, W.; Peng, W. Proof of reputation: A reputation-based consensus protocol for peer-to-peer network. In Proceedings of the International Conference on Database Systems for Advanced Applications, Gold Coast, Australia, 21–24 May 2018; pp. 666–681.
7. Nojoumian, M.; Golchubian, A.; Njilla, L.; Kwiat, K.; Kamhoua, C. Incentivizing blockchain miners to avoid dishonest mining strategies by a reputation-based paradigm. In Proceedings of the Computing Conference (CC), Berkeley, CA, USA, 16–18 July 2018; pp. 1118–1134.
8. Jadbabaie, A.; Lin, J.; Morse, A.S. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Trans. Autom. Control.* **2003**, *48*, 988–1001. [CrossRef]
9. Beard, R.W.; Stepanyan, V. Information consensus in distributed multiple vehicle coordinated control. In Proceedings of the 42nd IEEE International Conference on Decision and Control, (IEEE Cat. No. 03CH37475), Maui, HI, USA, 9–12 December 2003; Volume 2, pp. 2029–2034.
10. Bellingham, J.S.; Tillerson, M.; Alighanbari, M.; How, J.P. Cooperative path planning for multiple UAVs in dynamic and uncertain environments. In Proceedings of the 41st IEEE Conference on Decision and Control, Las Vegas, NV, USA, 10–13 December 2002; Volume 3, pp. 2816–2822.
11. Ren, W.; Beard, R.W. Decentralized scheme for spacecraft formation flying via the virtual structure approach. *J. Guid. Control. Dyn.* **2004**, *27*, 73–82. [CrossRef]
12. McLain, T.W.; Beard, R.W. Coordination variables, coordination functions, and cooperative timing missions. *J. Guid. Control. Dyn.* **2005**, *28*, 150–161. [CrossRef]
13. Zamir, L.; Shaan, A.; Nojoumian, M. ISRaft Consensus Algorithm for Autonomous Units. In Proceedings of the 29th International Conference on Network Protocols (ICNP), Dallas, TX, USA, 1–5 November 2021; pp. 1–6.
14. Bamakan, S.M.H.; Motavali, A.; Bondarti, A.B. A survey of blockchain consensus algorithms performance evaluation criteria. *Expert Syst. Appl.* **2020**, *154*, 113385. [CrossRef]
15. of Authority | Governance | Diverse Ecosystem, V.F.W.P. Vechain Whitepaper: Vechain Builders. Available online: https://www.vechain.org/whitepaper (accessed on 7 February 2022).
16. Castro, M.; Liskov, B. *Practical Byzantine Fault Tolerance*; OsDI: New Orleans, LA, USA, 1999; Volume 99, pp. 173–186.
17. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. In Proceedings of the 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), Philadelphia, PA, USA, 19–20 June 2014; pp. 305–319.
18. Nojoumian, M. Rational Trust Modeling. In Proceedings of the 9th Conference on Decision and Game Theory for Security (GameSec), Seattle, WA, USA, 29–31 October 2018; pp. 418–431.
19. Nojoumian, M. Trust, influence and reputation management based on human reasoning. In Proceedings of the 9th 4th AAAI Workshop on Incentives and Trust in E-Communities (WIT-EC), Austin, TX, USA, 25 January 2015; pp. 21–24.
20. Nojoumian, M.; Stinson, D.R.; Grainger, M. Unconditionally secure social secret sharing scheme. *Inf. Secur. Spec. Issue -Multi-Agent Distrib. Inf. Secur.* **2010**, *4*, 202–211. [CrossRef]
21. Nojoumian, M.; Golchubian, A.; Saputro, N.; Akkaya, K. Preventing collusion between SDN defenders and attackers using a game theoretical approach. In Proceedings of the Conference on Computer Communications Workshops (INFOCOM WKSHPS), Atlanta, GA, USA, 1–4 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 802–807.
22. Pourtahmasbi, P.; Nojoumian, M. Impacts of Trust Measurements on the Reputation-Based Mining Paradigm. In Proceedings of the 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), Paris, France, 27–30 September 2021; pp. 225–228.
23. Pourtahmasbi, P.; Nojoumian, M. Analysis of Reputation-Based Mining Paradigm Under Dishonest Mining Attacks. *Blockchain Res. Appl.* **2022**, *3*, 100065. [CrossRef]
24. Nojoumian, M.; Lethbridge, T.C. A new approach for the trust calculation in social networks. In Proceedings of the E-Business and Telecommunication Networks: 3rd International Conference on E-Business, Porto, Portugal, 26–28 July 2006; pp. 64–77.
25. Biswas, S.; Sharif, K.; Li, F.; Maharjan, S.; Mohanty, S.P.; Wang, Y. PoBT: A lightweight consensus algorithm for scalable IoT business blockchain. *IEEE Internet Things J.* **2019**, *7*, 2343–2355. [CrossRef]