SmartShards: Churn-Tolerant Continuously Available Distributed Ledger

Joseph Oglio, Mikhail Nesterenko$^{(\boxtimes)}$, and Gokarna Sharma

Department of Computer Science, Kent State University, Kent, OH 44242, USA
joglio@kent.edu, {mikhail,sharma}@cs.kent.edu

Abstract. We present *SmartShards*: a new sharding algorithm for improving Byzantine tolerance and churn resistance in blockchains. Our algorithm places a peer in multiple shards to create an overlap. This simplifies cross-shard communication and shard membership management. We describe *SmartShards*, prove it correct and evaluate its performance.

We propose several *SmartShards* extensions: defense against a slowly adaptive adversary, combining transactions into blocks, fortification against the join/leave attack.

AQ1

1 Introduction

Blockchain is a distributed digital ledger maintained by a network of independent peers. The technology provides transparency and immutability of records while promises decentralized control over the ledger. This facilitates cooperation among non-trusting entities: the peers agree on the records of the ledger. The peers usually do not belong to the same organization and may join and leave the network during its operation generating continuous churn. The peers themselves may not necessarily operate correctly. Instead, they may fail or even attack the network. Such behavior is modeled as Byzantine fault [20] where the faulty peer is allowed to behave arbitrarily.

Besides cryptocurrency [24,30], which generates a lot of recent public discourse, blockchain is used in a variety of applications: online auctions [19], marketplaces [29], supply chain [11], health care [3], Internet-of-Things [26], intellectual property rights [16], electric power industry [4] among many others.

Popular blockchains use proof-of-work based consensus algorithms [24] in which peers compete for the right to publish records on the blockchain by searching for solutions to cryptographic challenges. Such algorithms tend to be conceptually simple and robust. However, they are resource intensive and environmentally harmful [28]. Therefore, modern blockchain designs often focus on cooperative consensus algorithms. In these cooperative consensus algorithms, rather than compete, peers exchange messages to arrive at a joint decision.

Within many blockchain application domains, the size of the blockchain network needs to reach thousands and possibly hundreds of thousands of peers. This

necessitates systems which are highly scalable and resilient. However, traditional blockchain systems face significant scalability issues: to agree on a ledger record all peers have to communicate with each other. Hence, the communication cost grows with the network size.

Sharding is a technique that may potentially enhance blockchain scalability. In a sharded network, the peers are divided into groups called shards. Each shard processes transactions in parallel, improving overall system throughput. A number of sharded blockchain designs are presented in the literature. Refer to recent surveys [15,21,31] for an extensive review. Although promising, the scalability of shading blockchain design is hampered by the inherent features of a decentralized peer-to-peer systems. In particular, cross-shard transactions and churn.

In this paper we propose an approach of shard construction we call *SmartShards* that addresses both of these problems. *SmartShards* overlaps shards to improve shard communication coordination and membership.

Cross-Shard Transactions and Faults. A shard acts as a unit of either coordination or data ownership. For example, in Elastico [22], the shards concurrently verify different transactions. In *Rapidchain* [32], each shard stores data about a collection of wallets. A cross-shard transaction requires the coordination of multiple shards. For example, in *Rapidchain*, a transaction may be moving funds between wallets in two different shards. There are multiple ways to coordinate such transactions. Some blockchains use locking [17]. Other designs break the single transaction into sub-transactions to be executed separately. Alternatively, the system may have a dedicated reference committee that approves cross-shard transactions [22]. Regardless of the technique, the source and target shards have to communicate either directly or though intermediate coordinators.

Individual peers may be faulty. One approach to Byzantine fault-tolerance is to use cryptographic signatures. However, this may be computationally expensive or difficult to implement in a peer-to-peer system. Instead, we consider a classic approach where the correctness depends on sufficiently large majority of correct peers. To ensure information propagation correctness, multiple source shard peers have to broadcast the transaction data to multiple target shard peers. This limits the efficiency of cross-shard transaction processing. In *SmartShards* that we propose, each peer participates in multiple shards. The shard overlap is used to make inter-shard communication more efficient and robust.

Churn. Churn, continuous joining and leaving of peers, may disrupt the operation of a blockchain. In case of sharding, the problem is exacerbated since it may potentially lead to shard failure and structural network compromise. However, the problem of handling continuous churn within sharded blockchains remains underexplored.

In most systems [17,22,32], the churn is handled through periodic system re-configuration events. An epoch is the time of continuous system operation between such reconfiguration events. For such epoch-based systems, the rate of churn is considered low enough to ensure adequate system availability. If the churn is substantial, to ensure correct blockchain operation, the system has to

spend significant time re-configuring itself. This downtime increases with both the rate of churn and the scale of the system.

In a sharded blockchain, peers have to maintain both their shard membership and the links to the other shards for cross-shard transactions. In traditional systems, this information has to be maintained separately. In *SmartShards*, the peers are used for internal shard recording and for inter-shard communication. Therefore, the overlapping shards integrally maintain both membership and links. This simplifies churn handling in *SmartShards*.

Related Work. There are several blockchain designs that overlap coordination between consensus groups. *Monoxide* [27] is a competitive Proof-of-Work blockchain. To improve scalability, it uses multiple asynchronous consensus zones that maintain separate blockchains. These blockchains periodically synchronize by linking blocks across zones. A peer may work on any of these local blockchains. Thus, *Monoxide* has fixed zones with floating peer membership. *Hyperledger Fabric* [5] uses cooperative consensus. It also maintains multiple blockchains. It has a centralized ordering service for transaction synchronization. Peers may join and leave the network only when the system is off-line, i.e. outside an epoch.

Churn management in distributed systems has been extensively studied. Kuhn *et al.* [18] propose simulating a robust node by a collection of peers connected to peers of other simulated nodes. These simulated nodes retain there data despite individual peer churn. In this sense, these simulated nodes are similar to shards in blockchain networks. Foreback *et al.* [12,13] explore the theoretical limits of infinite churn, proposing techniques to maintain network connectivity despite continuous churn in general peer-to-peer networks.

Malkhi and Reiter [23] study Byzantine-robust quorum systems for distributed storage. A quorum is a collection of subsets of peers where each pair of subsets intersects. This arrangement is similar to *SmartShards*. However, in their approach, quorums are used to build a robust register and to synchronize the entire network and ensure consistent reading and writing of the shared data. In *SmartShards*, the quorums are used to maintain separate data in separate shards and only synchronize if the data is affected by smart-shard transactions.

SMARTCHAIN [8] is a blockchain algorithm that uses quorums to improve performance. However, *SMARTCHAIN* uses cryptographic signatures to verify cross-shard transactions. In contrast, *SmartShards* does not need the cryptographic signatures.

Baldoni *et al.* [7] use quorums for churn resistance in Byzantine fault-tolerant distributed storage. Their approach requires bounded churn and eventual system synchrony. *SmartShards* do not need such assumptions.

García-Pérez *et al.* [14] use overlapping quorums to achieve consensus despite Byzantine faults. Similar to *SmartShards*, the quorum intersections are used for data update synchronization. However, a "trust relationship" is required for quorum formation and maintenance. *SmartShards* do not need such trust relationship.

Our Contribution. *SmartShards* presented in this paper is a novel sharded approach to Byzantine tolerance and churn resistance in blockchain systems.

Given a basic Byzantine consensus algorithm that tolerates up to f faults, and with overlap between shards x, *SmartShards* operates correctly with at most f faults in each shard, and at most $\lfloor x/2 - 1 \rfloor$ faults in the overlap.

2   Notation and Problem Definition

System Model. We assume a peer-to-peer network. Each peer has a unique identifier. A peer may send a message to any other peer so long as it has the receiver's identifier. Communication is reliable. Peers communicate through authenticated channels: the receiver of the message may always identify the sender. Identity forging, such as Sybil attacks [10], is not considered.
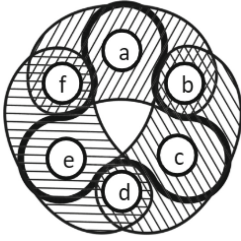
Fig. 1. *SmartShards* peer to shard allocation example. The shards are: $\{f, a, b\}$, $\{b, c, d\}$, $\{d, e, f\}$, $\{a, c, e\}$.
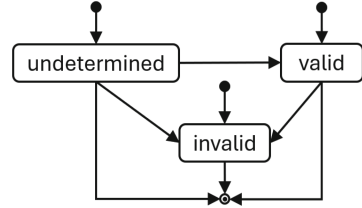
Fig. 2. Transaction validity transitions.

Shards, Transactions, Faults. A *shard* is a fully connected set of peers maintaining a blockchain. Shards A and B are considered *overlapping* if there exists at least one peer $p \in A \cap B$. For simplicity, we initially assume that a peer may belong to at most two shards. See example peer-to-shard allocation in Fig. 1.

The overlaps are used for communication between shards. We assume that the overlap size, x, is the same for all shards. Note that $x = 1$ in Fig. 4 and is not sufficient for fault tolerance. Two peers p and q that belong to the same shard are *shard mates*, or just *mates*. If q is a mate of p in some shard, then the *countershard* of p for q is the other shard to which q belongs.

A *wallet* is a means of storing funds. Each wallet has a unique identifier. A *client* is an entity that *owns* wallets. This ownership can be authenticated by the peers of the network. There may be multiple clients. A client submits transactions. A *transaction* is a transfer of funds from a *source* to a *target* wallet. For simplicity, we assume that each transaction has a single source and a single target wallet. A shard records transactions for a set of wallets. A *source shard* records the transaction if the source wallet is in this shard's set. Similarly, a *target shard* records a transaction if it maintains the target wallet. A client may have wallets in more than one shard.

There are two types of transactions. A *cross-shard transaction* moves funds between wallets of different shards. An *internal transaction* moves funds between

two wallets of the same shard. Note that for an internal transaction, the single shard is both the source and the target.

The funds are transferred in the form of *UTXOs* (unspent transaction outputs). A transaction is applied to an *input* UTXO and produces an *output* UTXO.

A ledger is a sequence of transactions recorded by a shard. Each peer in the shard maintains its own copy of this ledger. The peers are initialized with the same genesis ledger.

A transaction is *confirmed* by a peer if it is recorded in this peer's ledger. An output UTXO of a confirmed transaction is either spent or unspent. A UTXO is *unspent* if it is an output of a confirmed transaction and there is no other confirmed transaction that uses it as an input. It is *spent* if used as an input to a confirmed transaction. If a transaction is unconfirmed, its output UTXO is neither spent nor unspent.

A transaction, whether confirmed or not, is *valid* if it is applied to an unspent UTXO. A transaction is *invalid* if it is applied to a spent UTXO. Note that if the input of a transaction is neither spent nor unspent, then its validity is undetermined. The validity transitions of a transaction are, therefore, as follows, see Fig. 2. A submitted transaction may be either valid, invalid, or undetermined. An undetermined transaction may eventually become either valid or invalid. A transaction of determined validity may never become undetermined again. A valid transaction may become invalid if another transaction that spends its input UTXO is confirmed. A valid confirmed transaction is never invalidated. An invalid transaction, either confirmed or unconfirmed, never becomes valid.

A *Byzantine* peer is faulty. A faulty peer behaves arbitrarily. A *correct* peer is not faulty.

Consensus and Distributed Ledger Problems.

Definition 1. *In* the Consensus Problem, *every correct process is input a value v and must output an irrevocable decision subject to the following properties:*

CValidity: *if all correct peers are input the same value v, then every correct peer decides v;*
CAgreement: *no two correct peers decide differently;*
COrder: *any pair of decisions are ordered the same of every pair of peers;*
CLiveness: *every correct peer eventually decides.*

Tolerance threshold f is the maximum number of faulty processes. An algorithm solves the Consensus Problem if it satisfies the three properties above provided that the number of faults does not exceed f.

Definition 2. *In* the Distributed Ledger Problem, *peers confirm a particular transaction with the following three properties:*

LValidity: *every confirmed transaction is valid;*
LAgreement: *if a transaction is confirmed by one correct peer in a shard, it is confirmed by every correct peer in the shard; and*

LLiveness: *if a client submits a valid transaction, it is eventually either confirmed or invalidated.*

Algorithm 1: $SmartShards$, Variables

1 **Constants**
2 p ▷ peer id
3 $Shards$ ▷ set of all shard ids
4 $shard_1, shard_2 \in Shards$ ▷ ids of the shards this peer is in
5 $Peers_1, Peers_2$ ▷ sets of peer ids for $shard_1$ and $shard_2$
6 $CounterShards_1 \equiv \{\langle q, s \rangle \mid q \in Peers_1, s \in Shards\}$ ▷ countershards for $Peers_1$
7 $CounterShards_2 \equiv \{\langle q, s \rangle \mid q \in Peers_2, s \in Shards\}$ ▷ countershards for $Peers_1$

8 **Transaction variables**
9 $client$ ▷ client initiating the transaction
10 $sWallet$ ▷ source wallet
11 $tWallet$ ▷ target wallet
12 $UTXO$ ▷ source unspent transaction output
13 $sShard$ ▷ shard containing the source wallet
14 $tShard$ ▷ shard containing the target wallet

15 **Process variables**
16 $Ledger_1, Ledger_2$ ▷ sequence of committed transactions in $shard_1, shard_2$
        ▷ record format: $\langle UTXO, \ sShard, tShard, sWallet, tWallet \rangle$
17 $Transfers_1, Transfers_2$ ▷ set of cross-shard transaction receipts for $shard_1, shard_2$
      ▷ record format: $\langle UTXO, \ sShard, tShard, sWallet, tWallet, q \rangle$

---

3   *SmartShards* Description and Correctness Proof

Algorithm Outline. *SmartShards* uses a consensus algorithm *Consensus* to confirm transactions. The peers of the source shard execute *Consensus* to agree on the transaction. This is sufficient for an internal transaction where the source and target shards are the same. For a cross-shard transaction, the peers that overlap the source and target shard, inform the target shard of the transaction via a TRANSFER message. Once a peer of the target shard receives sufficient number of TRANSFER messages, it initiates consensus on the cross-shard transaction.

Constants and Variables Description. The constants and variables are listed in Algorithm 1. Let us start with constants. Each peer has a unique identifier p. The peer stores a set of all shard ids in $Shards$. Out of this set, p belongs to two shards: $shard_1$ and $shard_2$. The set $Peers_1$ contains the ids of the mates of p in $shard_1$. Similarly, $Peers_2$ are mates of p in $shard_2$. For each mate q of p, $CounterShards_1$ lists a set of countershards for all mates of p in $shard_1$. $Countershards_2$ is defined similarly for $shard_2$.

The variables maintained by p are of two kinds: transaction and process variables. Transaction variables pertain to a particular transaction: client id, source and target wallet, source UTXO, and source and target shard. Note that if the transaction is internal, then the source and target shards are the same.

Algorithm 2: *SmartShards*, Actions

---

18 **function** VALID$\langle client, UTXO, sShard \rangle$
19 $\quad$ ⊳ returns **true** if $UTXO$ is owned by $client$ and is unspent

20 ⊳ actions for $shard_1$
21 **upon receive** TRANSACTION $\langle client, UTXO, sShard, tShard, sWallet, tWallet \rangle$
22 $\quad$ **if** VALID$(client, UTXO, sShard)$ and $sShard = shard_1$ **then**
23 $\quad\quad$ STARTCONSENSUSTX$(UTXO, sShard, tShard, sWallet, tWallet)$

24 **upon** ENDCONSENSUSTX$\langle UTXO, sShard, tShard, sWallet, tWallet \rangle$
25 $\quad$ **if** VALID$(client, UTXO, sShard)$ **then**
26 $\quad\quad$ **append** $\langle UTXO, sShard, tShard, sWallet, tWallet \rangle$ to $Ledger_1$
27 $\quad\quad$ **if** $sShard \neq tShard$ and $tShard = shard_2$ **then**
28 $\quad\quad\quad$ **send** TRANSFER $(UTXO, sShard, tShard, sWallet, tWallet)$ to $tShard$
29 $\quad\quad$ **else**
30 $\quad\quad\quad$ **send** TXCONFIRMATION $\langle p, UTXO \rangle$ to $client$

31 **upon** ENDCONSENSUSTR$\langle UTXO, sShard, tShard, sWallet, tWallet, result \rangle$
32 $\quad$ **if** $result$ **then**
33 $\quad\quad$ **append** $\langle UTXO, sShard, tShard, sWallet, tWallet \rangle$ to $Ledger_1$
34 $\quad\quad$ **send** TXCONFIRMATION $\langle p, UTXO \rangle$ to $client$

35 **upon receive** TRANSFER$\langle UTXO, fromShard, toShard, sWallet, tWallet \rangle$ from q
36 $\quad$ **if** $q \in Peers_1$ and $CounterShards_1.q = fromShard$ and
37 $\quad\quad$ $\langle UTXO, fromShard, toShard, sWallet, tWallet, q \rangle \notin Transfers_1$ and
38 $\quad\quad$ $toShard = shard_1$ **then**
39 $\quad\quad$ **add** $\langle UTXO, fromShard, toShard, sWallet, tWallet, q \rangle$ to $Transfers_1$
40 $\quad\quad$ **if** $\#\{\langle UTXO, fromShard, toShard, sWallet, tWallet, * \rangle \in$
$\quad\quad\quad Transfers_1\} \geq \lceil \#\{\langle q, s \rangle \in CounterShards_1 : s = sShard\}/2 \rceil$ and
$\quad\quad\quad \langle UTXO, fromShard, toShard, sWallet, tWallet \rangle \notin Ledger_1$ **then**
41 $\quad\quad\quad$ STARTCONSENSUSTR$(UTXO, fromShard, toShard, sWallet, tWallet, \textbf{true})$

42 ⊳ actions for $shard_2$ similar

In addition to transaction variables, p maintains process variables in all transactions. Specifically, $Ledger_1$ contains records of confirmed transactions for $shard_1$. Variable $Ledger_2$ keeps the records for $shard_2$. Variable $Transfers_1$ collects notifications of confirmed transactions from the source shard peers.

Actions Description. *SmartShard* actions are listed in Algorithm 2. The function VALID is used in the actions of *SmartShards*. It examines the shard ledger of p to check if the specific $UTXO$ is in the client wallet and is unspent. That is, this $UTXO$ is not an input to another transaction in this ledger.

There are four actions in the algorithm: TRANSACTION, ENDCONSENSUSTX, ENDCONSENSUSTR, and TRANSFER. The TRANSACTION action, see Line 21, processes client transaction receipts in the source shard. If a transaction is valid, the peer starts the consensus algorithm for this transaction.

Once this consensus is done, the ENDCONSENSUSTX action, see Line 24, at the source shard checks if the transaction is still valid and confirms the transaction by recording it in the local ledger and sends a confirmation message to the client. If the transaction is cross-shard, and the peer is in both source and target shards, then the peer sends a TRANSFER message to the target shard notifying it of the successful confirmation.

Algorithm 3: *SmartShards*, Client Actions and Variables

---

43 Constants
44 $client$ ▷ client id
45 $UTXO$
46 f ▷ consensus tolerance threshold

47 Variables
48 $confirmed$ ▷ status of the pending transaction
49 $SMSmembers$ ▷ set of sets of peers, history of shard membership
50 $Confirmations$ ▷ set of peers that confirmed the transaction

51 Actions
52 upon SUBMITTX⟨$UTXO$⟩
53 $confirmed = $ **false**
54   **while** $\neg confirmed$ **do**   ▷ periodically resend transaction request
55   **send** TRANSACTION⟨$client, UTXO$⟩ to SMS

56 upon receive MEMBERSHIP ⟨$Members$⟩ from SMS
57 $SMSmembers := \{Members\} \cup SMSmembers$

58 upon receive TXCONFIRMATION ⟨$peer, UTXO$⟩ from $peer$
59 $Confirmations := Confirmations \cup peer$
60   **if** $\exists\ Members \in SMSmembers : \#\{Members \cap Confirmations\} > f$ **then**
61 $confirmed := $ **true**

62 ▷ join and leave actions similar

The TRANSFER action handles the TRANSFER message receipt, see Line 35. A recipient peer p in $shard_1$ collects such receipts in $Transfers_1$. If p receives TRANSFER messages from greater than half of the peers in the overlap between the source and target shards, it considers the transaction to be valid and then p initiates consensus in the target shard on the cross-shard transaction. Every peer in the target shard participates in this consensus. Each peer uses the number of received TRANSFER messages to set its initial value for consensus. If the number of received TRANSFER messages is greater than half the number of peers in the overlap then the value is set to **true** otherwise **false**.

Once consensus on a transfer is reached, the ENDCONSENSUSTR action, see Line 31, at the target shard checks whether the peers agree on the validity of the transaction. If agreement is reached, and $result$, the consensus value, is **true**, the transaction is recorded in the ledger and sends a confirmation message to the client.

Correctness Proof.

Lemma 1. SmartShards *satisfies the LValidity property of the Distributed Ledger Problem.*

Proof. To satisfy LValidity, we need to show that only valid transactions are confirmed. Let us consider internal transactions first. In this case, the source and target shards are the same. Therefore, *SmartShards* executes only the TRANSACTION and ENDCONSENSUSTX actions without transmitting any TRANSFER messages. The peers of the shard execute the *Consensus* algorithm for the transaction within a single shard. An internal transaction is only confirmed by the

ENDCONSENSUSTX action. Before confirming, the action checks the transaction validity. That is, only valid transactions are confirmed. Valid confirmed transactions may not be invalidated.

Let us now deal with cross-shard transactions. Such transactions are confirmed in the source and target shards separately. By an argument similar to the above, we can show that a cross-shard transaction is confirmed in the source shard only if it is valid. If such a transaction is confirmed in the source shard, the source shard peers send a TRANSFER message. This initiates consensus in the target shard. A correct peer starts such consensus only after it receives a TRANSFER message from more than half of the peers in the overlap between the source and target shards. This means that at least one correct peer in the source shard sends such message. That is, this transaction is valid.

Note that a cross-shard transaction is confirmed by the ENDCONSENSUSTR action in the target shard. If this consensus is started by a correct peer, the transaction is valid. Alternatively, a Byzantine peer may start *Consensus* on a potentially invalid transaction. For such a transaction, correct peers do not have a sufficient number of TRANSFER messages. Therefore, all correct peers are input an initial value of **false**. By the CValidity property of consensus, if all correct peers are input **false**, they must decide **false**. In which case, the transaction is not confirmed. Thus, *SmartShards* satisfies LValidity for both internal and cross-shard transactions.   □

Lemma 2. SmartShards *satisfies the LAgreement property of the Distributed Ledger Problem.*

Proof. We prove this property by induction. Each peer starts with the same genesis ledger which is assumed to contain valid transactions. Let the ledger at each peer contain i valid transactions. Any transaction is confirmed by either the execution of the ENDCONSENSUSTX or ENDCONSENSUSTR action. Let us consider the action for transaction $i + 1$. By COrder property, the order of ENDCONSENSUSTX and ENDCONSENSUSTR actions are the same for every peer. By CAgreement, this is the same transaction at every peer. Since the ledger up to the i-th transaction is the same for each peer, the validity of this transaction is also going to be the same. That is, this transaction is going to be either confirmed or rejected by all correct peers. Thus, the ledger remains consistent after processing $i + 1$ transactions.   □

Lemma 3. SmartShards *satisfies the LLiveness property of the Distributed Ledger Problem.*

Proof. For internal transactions, the peers are running *Consensus*. Therefore, LLiveness is satisfied because the CLivenss property of *Consensus* ensures that every transaction is eventually either confirmed or invalidated.

Let us discuss cross-shard transactions. Similarly to internal transactions, the peers of the source shard are running *Consensus*. Therefore, they eventually either confirm or invalidate any cross-shard transaction. If the transaction is confirmed by the source shard, the source shard peers that overlap the target

shard send TRANSFER messages. Greater than half of such peers are assumed to be correct. Therefore, each target peer eventually gets more than half such TRANSFER messages.

If a target peer gets more than half TRANSFER messages, it will start *Consensus*. If all correct peers received TRANSFER messages from more than half of the overlap, then they all use *true* as the input for *Consensus*. By CValidity, this consensus must succeed. Therefore, LLiveness holds for cross-shard transactions as well. Hence the lemma.   □

The below theorem follows from Lemmas 1, 2, and 3. Recall that x is the shard overlap size.

Theorem 1. *Algorithm* SmartShards *solves the Distributed Ledger Problem with at most f Byzantine faults in each shard and at most $\lfloor x/2 - 1 \rfloor$ faults in the overlap between shards.*

Fault tolerance threshold estimation. Let us address the relation between x and f. From the above theorem it follows that $f < x/2$. However, we may be able to get a more optimistic bound. In classic sharding algorithms, it is assumed that faults are distributed across shards somewhat evenly. To put another way, the number of faults in a shard is proportional to shard size. We apply this reasoning to the shard overlap size.

Let s be the number of shards in the network and z be the shard size. Assume the shard fault tolerance threshold is $f < z/3$. Since every pair of shards overlap over x peers, the shard size is $z = x(s-1)$. If the number of faults is proportional to overlap size then the tolerance threshold can be estimated as follows:

$$f < \frac{z}{3} = \frac{x(s-1)}{3}.$$

Churn Tolerance. SmartShards may be modified to tolerate churn: peer joining and leaving. We consider only cooperative churn where joining and leaving processes are correct [13]. Faulty churning peers are assumed to be within the general tolerance thresholds f and x. There are two aspects: (i) the client needs to ascertain that the appropriate member peers confirm the transaction and (ii) the peers themselves need to agree on the current shard membership.

We assume the existence of the Shard Membership Service (SMS) – a trusted service providing membership information to the clients. A client submits its transaction to the SMS. The SMS forwards this transaction to the peers in the source shard and replies to the client with the current set of peer identifiers in this shard. Once the peers receive the transaction from the SMS, they run the consensus algorithm and send the confirmation to the client.

A client may re-send the transaction. In this case, the SMS behaves similarly: it forwards the transaction to the peers and sends the membership set to the client. Due to peer churn, the membership set may differ every time. If the peers receive a request for a transaction that is already confirmed, they re-send the confirmation to the client.

Potentially, the SMS may return the exact set of peers that handle the transaction. However, such service may be difficult to implement as it needs to synchronize its operation with peers joining and leaving. Instead, we consider a weaker SMS assumption. If the client submits infinitely many requests for a specific valid transaction, then the SMS provides infinitely many sets containing at least $f + 1$ peers that reply to the transaction request. *Weak SMS* is the SMS that satisfies this assumption. Henceforth, we refer to weak SMS as just SMS.

The client actions are shown in Algorithm 3. We show the code for a single transaction confirmation. Until confirmed, the client periodically sends the transaction requests to the SMS. The client collects membership sets sent back by the SMS in $SMSmembers$. The client also collects peer confirmations in $Confirmations$. The client considers the transaction confirmed if it gets at least $f + 1$ confirmations for this transaction from one of the membership sets received from the SMS.

Let us now discuss churn. A peer wishing to join or leave *SmartShards* submits a transaction with this request. A joining peer utilizes the SMS. The process is similar to a client submitting a regular UTXO-related transaction. With their confirmation, the peers in the shard send their complete ledger to the joining peer. Once this peer receives $f + 1$ confirmations from one of the shard membership sets supplied by the SMS, the peer is ready to participate in *SmartShards*. The algorithm for a leaving peer is simpler since it knows the shard membership. The leaving peer just submits a leave request to the members of its shard. It may leave the shard once it receives $f + 1$ confirmations. After the peer leaves, it may not receive or send any messages. We summarize this discussion in the below theorem.

Theorem 2. *Algorithm* SmartShards *with weak SMS solves the Distributed Ledger Problem with at most f Byzantine faults in each shard and at most* $\lfloor x/2 - 1 \rfloor$ *faults in the overlap between shards despite peer churn.*

4 Performance Evaluation

Simulation Setup. We evaluate the performance of *SmartShards* in the abstract algorithm simulator QUANTAS [25]. The QUANTAS code for the *SmartShards* simulation as well as our performance evaluation data is available online [1,2].

The simulated network consists of individual peers. Each pair of peers communicate via a message-passing channel. The channels are FIFO and reliable. A single computation is modeled as a sequence of rounds. In each round, a peer receives messages that were sent to it in the previous rounds, performs local computation, and sends messages to the other peers.

The receipt of each message may be delayed up to some maximum amount which various by experiment. The network is modeled with n peers divided into s shards.
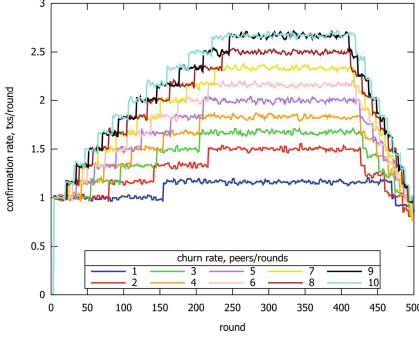
Fig. 3. Transaction confirmation rate timing diagram. Peers request to join the network every round from 1 to 250. They request to leave the network every round from 251 to 500.
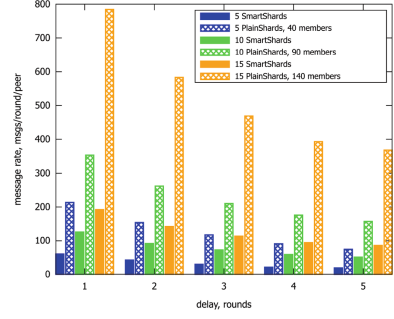
Fig. 4. Message rates for varied shard sizes and max message delay for the *SmartShards* algorithm vs. *PlainShards*.

We do not explicitly simulate clients or faults. Each shard processes transactions independently. We evaluate *SmartShards* where peers participate in multiple overlapping shards against *PlainShards* where each peer is in at most one shard. In *SmartShards* each peer is in at most two shards. If two shards overlap, x is the number of peers in this overlap. Therefore, total number of peers in the network is subject to this formula:

$$n = \frac{s(s-1)x}{2}$$

In *SmartShards*, each shard is running *PBFT*. *PBFT* [9] is a common leader-based consensus algorithm. Transactions are proposed by the *PBFT* leader of each shard. All transactions are valid and have a chance of being cross-shard that is governed by the number of shards in the network.

Peers in the network may join and leave shards. Unless otherwise stated, in each round, a single peer requests to join and a single peer requests to leave the network.

SmartShards and *PlainShards* differ in their implementation of cross-shard transactions and transaction membership. Since there is no shard overlap, in case of a cross-shard transaction, each source shard peer informs every target peer of source shard consensus decision. Also, each peer has to maintain memberships of all network shards. Therefore, once a peer joins or leaves the network, it has to inform all network peers.

We simulate computations of 500 rounds. Unless otherwise stated, each data point is an average of 10 tests.

Experiment Description. In the first experiment we observed the dynamics of *SmartShards* churn handling. The results of a representative computations are shown in Fig. 3. In every round from 1 to 250, new peers request to join the

network. Then, in rounds 251 though 500, randomly selected peers request to leave the network. *PBFT* leaders and peers that have not joined yet, are not selected to leave. If there are least 25 peers in the shard overlap between each shard on average, a new shard is created, random peers from donor shards join a new shard. If there are less than 9 peers in the shard overlap between each shard on average, a shard is destroyed and its peers are randomly distributed to other shards. We plot a rolling average of transaction confirmation rate over 100 tests with a window of 3 rounds. We vary the churn rate: the number of peers requesting to join or leave the network every round.

Initially, the network contains 100 peers divided into 5 shards with an overlap of 10. The confirmation rate reflects the number of available shards: the greater the number of shards the higher the confirmation rate. The experiments show the robustness of *SmartShards* with the respect to churn.

Figure 4 compares *SmartShards* with *PlainShards*, the non-overlapping sharded implementation. We compare the message exchange rate for *SmartShards* and *PlainShards* networks of comparable sizes. The message rate is the total number of messages exchanged divided by the number of rounds in the computation and the number of peers in the network. The figure indicates that *SmartShards* outperforms *PlainShards* due to more efficient implementation of cross-shard transaction and churn handling. Each *SmartShards* experiment uses an overlap of 10.

For the next set of experiments, we vary message delay and observe its influence on network performance. In Figs. 5, 7, and 9, we change the number of shards in *SmartShards* and observe the dynamics of throughput, latency and message rate respectively. Throughput is computed as the total number of transactions confirmed during the computation. Latency is the number of rounds that elapse from when a transaction is submitted until it is confirmed by *PBFT*. The figures indicate that with the increase of the number of shards the performance of *SmartShards* improves with a relatively modest increase in message expense.

In Figs. 6, 8, and 10 we change the intersection size and observe its influence on throughput, latency and message rate. Figures 6 and 8 demonstrate that as the number of intersections between shards in the network increases, the throughput and latency remain unaffected. However, Fig. 10 shows that the increase in the number of intersections increases the message rate as expected due to the increase in shard sizes. This cost is relatively low due to the benefit of increasing the Byzantine tolerance threshold of the algorithm.

5 Extensions and Implementation Considerations

Multiple Sources and Targets Transactions. *SmartShards* can be adapted to handle a *multi-transaction* that transfers funds from multiple source wallets or to multiple target wallets atomically. If the wallets belong to the same shard, the modification is simple since the peers of the same shard confirm the transaction. However, it becomes complicated if the wallets are located in different shards.

In this case, the multi-transaction is preceded by the Consolidation Phase and followed by the Distribution Phase. The actual transaction is carried out in a *consolidation shard*. This shard may be one of the source or target shards or an unrelated shard. The *Consolidation Phase* consists of moving the source UTXOs into the consolidation shard. Each source UTXO is transferred to the consolidation shard without changing ownership of the UTXO. That is, the same client owns the wallet which contained the UTXO in the source as in the consolidation shard. After the Consolidation Phase is complete, *SmartShards* confirms the multi-transaction in a single consolidation shard. After this execution, the target UTXOs are transferred to the target shards during the *Distribution Phase*. Again, there is no ownership change during such a transfer.
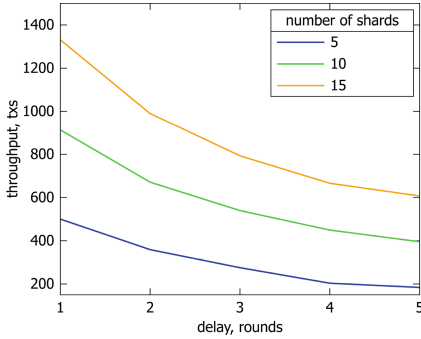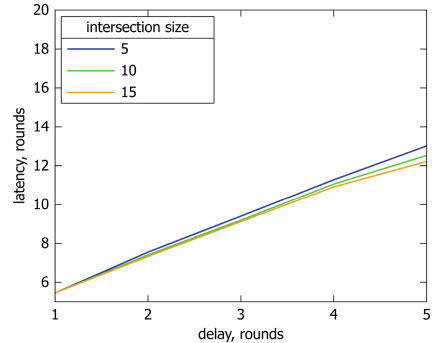
Fig. 5. Number of approved transactions depending on the message delay for various number of shards.
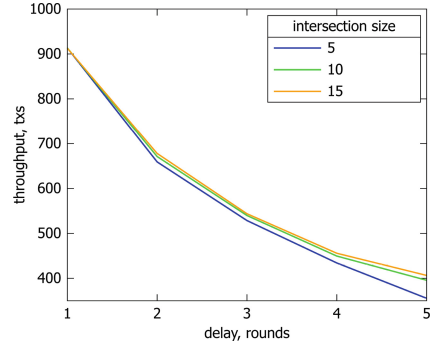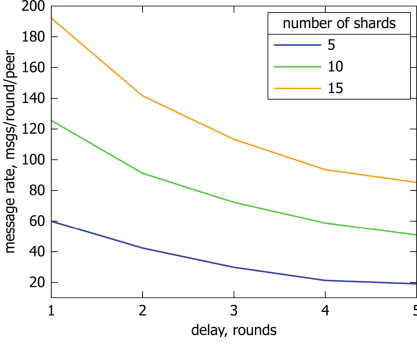

Fig. 6. Number of approved transactions depending on the message delay for various intersection sizes.
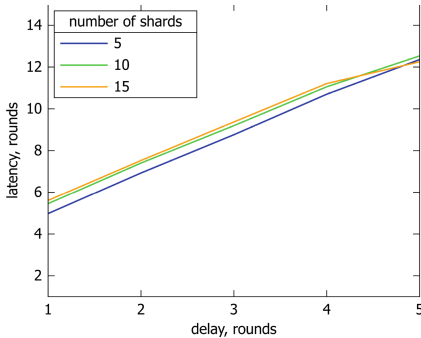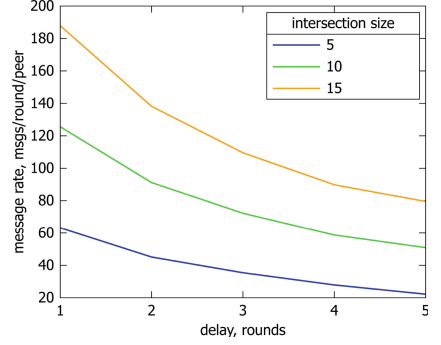

Fig. 7. Latency of approving a transaction depending on the message delay for various number of shards.

Fig. 8. Latency of approving a transaction depending on the message delay for various intersection sizes.

Fig. 9. Message exchange rate depending on message delay.

Fig. 10. Average message rate for varied intersection sizes and max message delay.

Defense Against Membership Attacks. *SmartShards* may be fortified against a *slowly adaptive adversary* [32] that requires a certain time period to corrupt new peers. Such an adversary may focus on compromising a single shard or a single shard intersection. If the corruption period is known, peers are ejected from their shards before a timer expires to prevent a Byzantine majority from compromising the system. Such peers then join a different shard shuffling the peers and distributing the faults.

Similarly, defense against a leave/join attack may be incorporated into *SmartShards*. In such an attack, the faulty nodes repeatedly join and leave the network hoping to get into the same shard in an attempt to exceed its tolerance threshold. To counteract it, a Cuckoo rule may be applied [6]. If a node joins a shard, another arbitrary node has to leave this shard and join another shard. Thus, the adversary loses control over fault node shard selection.

Implementation Considerations. As with any blockchain, to improve performance, multiple transactions may be confirmed in a block in *SmartShards*. Thus, a single round of agreement is required for confirmation of all transactions in the block. In this case, the peers of the shard confirm several transactions as a block and then carry out the transfer part of each transaction individually. The block may contain regular UTXO transactions as well as peer join and leave transactions.

Let us comment on the implementation of the Shard Membership Service (SMS) component of *SmartShards*, which clients use to keep track of membership changes. Effectively, the SMS is a trusted source of membership information. As such, it may be implemented as a side-chain of membership records, a bootstrap service, or a Byzantine-robust host that reliably answers client membership queries.

6   Concluding Remarks

In this paper we explored the idea of overlapping shards. We showed how it can be used to improve efficiency and robustness of a sharded blockchain. In the future, it would be interesting to implement *SmartShards* in a real pear-to-peer system and compare it against traditional non-overlapping sharding blockchains. Alternatively, *SmartShards* may be used directly in existing blockchains to overlap shards and enhance their fault-tolerance and functional performance.

References

1. Smartshards implementation. https://github.com/QuantasSupport/Quantas/tree/SmartShards
2. Smartshards perfromance evaluation data. http://www.cs.kent.edu/mikhail/Research/trail.output.tar.gz Feb 2025
3. Agbo, C.C., Mahmoud, Q.H., Eklund, J.M.: Blockchain technology in healthcare: a systematic review. In: Healthcare, vol. 7, p. 56. MDPI (2019)
4. Andoni, M., et al.: Blockchain technology in the energy sector: a systematic review of challenges and opportunities. Renew. Sustain. Energy Rev. **100**, 143–174 (2019)
5. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, pp. 1–15 (2018)
6. Awerbuch, B., Scheideler, C.: Towards a scalable and robust DHT. In: Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 318–327 (2006)
7. Baldoni, R., Bonomi, S., Nezhad, A.S.: A protocol for implementing byzantine storage in churn-prone distributed systems. Theor. Comput. Sci. **512**, 28–40 (2013)
8. Bessani, A., Alchieri, E., Sousa, J., Oliveira, A., Pedone, F.: From byzantine replication to blockchain: Consensus is only the beginning. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 424–436. IEEE (2020)
9. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
10. Douceur, J.R.: The sybil attack. In: International workshop on peer-to-peer systems, pp. 251–260. Springer (2002)
11. Dujak, D., Sajter, D.: Blockchain applications in supply chain. SMART supply network, pp. 21–46 (2019)
12. Foreback, D., Nesterenko, M., Tixeuil, S.: Infinite unlimited churn. arXiv preprint arXiv:1608.00726 (2016)
13. Foreback, D., Nesterenko, M., Tixeuil, S.: Churn possibilities and impossibilities. In: Networked Systems: 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9–11, 2018, Revised Selected Papers 6, pp. 303–317. Springer (2019)
14. García-Pérez, A., Gotsman, A.: Federated byzantine quorum systems. In: 22nd International Conference on Principles of Distributed Systems (OPODIS 2018). Schloss-Dagstuhl-Leibniz Zentrum für Informatik, (2019)
15. Hafid, A., Hafid, A.S., Samih, M.: Scaling blockchains: a comprehensive survey. In: IEEE Access **8**, 125244–125262 (2020)

16. Ito, K., O'Dair, M.: A critical examination of the application of blockchain technology to intellectual property management. Business Transform. Blockchain **II**, 317–335 (2019)
17. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E.,Ford, B.: Omniledger: aA secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 583–598. IEEE (2018)
18. Kuhn, F., Schmid, S., Wattenhofer, R.: Towards worst-case churn resistant peer-to-peer systems. Distrib. Comput. **22**, 249–267 (2010)
19. Lafourcade, P., Nopere, M., Picot, J., Pizzuti, D., Roudeix, E.: Security analysis of auctionity: a blockchain based e-auction. In: Foundations and Practice of Security: 12th International Symposium, FPS 2019, Toulouse, France, November 5–7, 2019, Revised Selected Papers 12, pp. 290–307. Springer, 2020 ¡error l="308" c="Invalid command: paragraph not started." /¿
20. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
21. Liu, X., Xie, H., Yan, Z., Liang, X.: A survey on blockchain sharding. ISA Trans. **141**, 30–43 (2023)
22. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 17–30 (2016)
23. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Comput. **11**(4), 203–213 (1998)
24. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
25. Oglio, J., Hood, K., Nesterenko, M., Tixeuil, S.: Quantas: quantitative user-friendly adaptable networked things abstract simulator. In: Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems, pp. 40–46 (2022)
26. Samaniego, M., Jamsrandorj, U., Deters, R.: Blockchain as a service for Iot. In: 2016 IEEE International Conference on Internet of Things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData), pp. 433–436. IEEE (2016)
27. Wang, J., Wang, H.: Monoxide: scale out blockchains with asynchronous consensus zones. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp. 95–112 (2019)
28. Wendl, M., Doan, M.H., Sassen, R.: The environmental impact of cryptocurrencies using proof of work and proof of stake consensus algorithms: a systematic review. J. Environ. Manage. **326**, 116530 (2023)
29. White, B., Mahanti, A., Passi, K.: Characterizing the OpenSea NFT marketplace. In: Companion Proceedings of the Web Conference 2022, pp. 488–496 (2022)
30. Wood, G.: Ethereum: a secure decentralized generalized transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)
31. Yu, G., Wang, X., Yu, K., Ni, W., Zhang, J.A., Liu, R.P.: Sharding in blockchains: Survey. IEEE Access **8**, 14155–14181 (2020)
32. Zamani, M., Movahedi, M., Raykova, M: Rapidchain: scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 931–948 (2018)