Bare Demo of IEEEtran.cls for IEEE Conferences

Michael Shell
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: http://www.michaelshell.org/contact.html

Homer Simpson
Twentieth Century Fox
Springfield, USA
Email: homer@thesimpsons.com

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678–2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—The abstract goes here.**

I. Introduction

II. Background and Related Work

Making early blockchain systems scalable has received much attention from both the academia and the industry in recent years. This section gives an mini taxonomy of the state-of-the-art, categorised by the scalability approach. At the end of this section, we describe the important–asynchronous subset consensus (ACS), which is the key building block of our work.

A. *Off-chain transactions and payment networks*

Off-chain transactions make use of the following fact. If nodes make frequent transactions, then it is not necessary to store every transaction on the blockchain, only the net settlement is necessary. The best examples are Lightning Network [1] and Duplex Micropayment Channels [2].

Off-chain transactions promises significant scalability improvements, but it suffer from the problem of Bitcoin. Namely, proof-of-work consumes an unreasonable amount of power. Further, payment channels complicate user experience. As we mentioned earlier, each node must deposit some Bitcoins into a multi-signature account. The users must pick a suitable amount. If the deposit is too low it would not allow large transactions. If the deposit is too high the user is locked out of much of its Bitcoins for use outside of the channel. In addition, the user must proactively check whether the counterparty has broadcasted an old channel state so that the user does not lose Bitcoins. Payment channel, in theory, solves the scalability problem of early blockchain systems, but to the best of our knowledge, its exact scalability characteristics are not investigated.

B. *Permissioned systems based on Byzantine consensus*

This category of systems uses traditional Byzantine consensus algorithms such as PBFT [3]. In essence, they contain a fixed set of nodes (sometimes called validating peer) that run a Byzantine consensus algorithm to decide on new blocks. This is often used in permissioned system where the validators must be predetermined, for example, Hyperledger Fabric [4].

A nice aspect of Byzantine consensus and in particular PBFT is that it can handle much more transactions than classical blockchain systems. PBFT can, for example, achieve 10,000 TX/s if the number of validating peer is under 16 [5, Section 5.2]. Further, in contrast to proof-of-work, PBFT consensus is final. That is, transaction history cannot be re-written if it is already in the blockchain.

The major drawback of Byzantine consensus based systems is that it does not scale in terms of the number of validating peers. Going back to PBFT, its transaction rate drops to under 5000 TX/s when the number of validating peer is 64 [5, Section 5.2]. Moreover, the validating peers are predetermined which makes the system unsuitable for the open internet.

C. *Combining proof-of-work with Byzantine consensus*

Recent research has developed a class of hybrid systems which uses PoW for committee election, and Byzantine consensus algorithms to agree on transactions. This design is primarily for permissionless system because the PoW leader election aspect prevents Sybils. Some examples are SCP [6], ByzCoin [7] and Solidus [8].

This technique overcomes the early blockchain scalability issue by delegating the transaction validation to the Byzantine consensus protocol (e.g. PBFT in ByzCoin [7]). A major trade-off of such systems is that they cannot guarantee correctness when there is a large number of malicious nodes (but less than a majority). For SCP, ByzCoin and Solidus, they all have some probability to elect more than t Byzantine nodes into the committee, where t is typically just under a third of the committee size (a lower bound of Byzantine consensus [9]). This problem is especially difficult to solve because the committee is always much smaller than the population size which has more than t Byzantine nodes. Classical blockchain does not have this problem because they do not use Byzantine consensus. Further, due to the fact that these systems must reach consensus on all transactions, none of them achieves horizontal scalability.

D. *Sharding*

Sharding is a technique to achieve horizontal scalability by grouping nodes into multiple committees or shards. Nodes within a single shard run a Byzantine consensus algorithm to agree on a set of transactions that belong to that specific shard. An intra-shard protocol is needed for transactions that involve nodes from more than one shard. The number of shards grows linearly with respect to the total computational power in the network. This scheme achieves horizontal scalability because if every shard commits transactions at the same throughput, then adding more shard would naturally result in a linear increase of global throughput. Examples of blockchain systems that use sharding are Elastico [10] and OmniLedger [11].

The biggest limitation of sharding is that it is only optimal is transactions stay in the same shard. In fact, Elastico cannot atomically process inter-shard transactions. OmniLedger has an inter-shard transaction protocol but choosing a good shard size is difficult. A large shard size would make the system less scalable because the Byzantine consensus algorithm must be run by a large number of nodes. A small shard size would result in a large number of inter-shard transactions which also hinder scalability. The authors of OmniLedger noted that inter-shard transactions have significantly higher latency compared to the consensus protocol [11]. Furthermore, since no shards can be compromised for the system to function correctly, the adversaries have more opportunities to compromise the system.

E. Asynchronous subset consensus

ACS is an especially useful primitive for blockchain systems. It allows any party to propose a value and the result is the set union of all the proposed values by the majority. Concretely, ACS needs to satisfy the following properties (adapted from [5]).

Definition 1. Asynchronous subset consensus

There are n nodes, of which at most t might experience Byzantine fault. Node i starts with a non-empty set of input values C_i. The nodes must decide an output C, satisfying the following.

1) Agreement: *If a correct node outputs C, then every node outputs C.*
2) Validity: *If any correct node outputs a set C, then $|C| \geq n - t$ and C contains the input of at least $n - 2t$ nodes.*
3) Termination: *If $n - t$ nodes receive an input, then all correct nodes produce an output.*

ACS has the nice property of censorship resilience when compared to other consensus algorithms. For instance, a leader is elected in PBFT [3], if the leader is malicious but follows the protocol, then it can selectively filter transactions. In contrast, every party in ACS are involved in the proposal phase, and it guarantees that if $n - 2t$ parties propose the same transaction, then it must be in the agreed output. Thus, if some value is submitted to at least $n - 2t$ nodes, it is guaranteed to be in the consensus result. The total communication complexity of ACS is $O(n^2|v| + \lambda n^3 \log(n))$, where n is the number of nodes, $|v|$ is the message size and λ is the security parameter. For a detailed description of ACS, we refer to the HoneyBadgerBFT work [5].

III. System Architecture

Our system consists of three protocols—the consensus protocol, the transaction protocol and the validation protocol. It is based on our prior work on TrustChain [12], where every node independently interacts with each other via their own blockchain[1].

A. Overview and intuition

Unlike most blockchain systems, we do not use a global ledger. Instead, every node has their own hash chain. The nodes only store transactions (TX) that they are involved in on their hash chain. We introduce a special block called checkpoint (CP) block, which represents the state of a hash chain. The data structure is called Extended TrustChain, a visualisation can be seen on Figure 1.

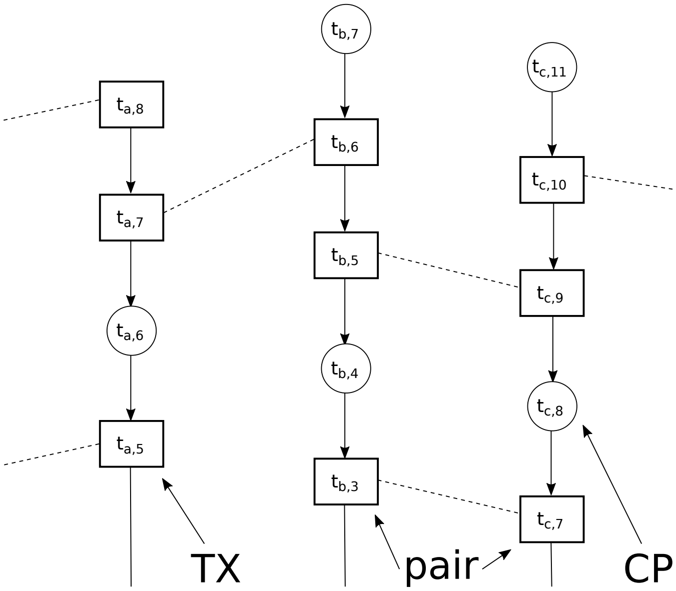

Fig. 1: Visualisation of the Extended TrustChain. $t_{u,i}$ represents a TX block on u's chain with a sequence number of i. $c_{v,j}$ represents a CP block on v's chain with a sequence number of j.

We randomly elect special nodes called facilitators in every round. The facilitators reach consensus on CP blocks and the consensus result is essentially a set of CP blocks of from (nearly) all the nodes. While this gives us consensus on a representation of the global state, it does not imply consensus on transactions. To this end, we introduce the validation protocol that allows any nodes to challenge any other node to produce a set of TX blocks that computes to the CP block in consensus. Hence we implicitly achieve consensus on transactions.

B. Model and assumptions

We assume purely asynchronous channels with eventual delivery. Thus in no stage of our protocol do we make timing

[1] It is originally called MultiChain but the name has changed to TrustChain

assumptions. The adversary has full control of the delivery schedule and the message ordering of all messages. But they are not allowed to drop messages except for their own[2]. Further, we assume there exists a Public Key Infrastructure (PKI), and nodes are identified by their unique and permanent public key.

In our model, we consider N nodes, which is the population size. n of them are facilitators, t out of n are malicious and the inequality $n \geq 3t + 1$ must hold. This is from the work of Pease, Shostak and Lamport, where they show a network of n nodes cannot reach Byzantine agreement with $t \geq n/3$ [9]. Further, the inequality $N \geq n + t$ must also hold. This is due to our system design, which becomes clear in Section III-D2.

Our threat model is as follows. We use a restricted version of the adaptive corruption model. The first restriction is that corrupted node can only change across rounds. That is, if a round has started, the corrupted nodes cannot change until the next round. The second restriction is that the adversary, presumably controlling all the corrupted nodes, is forgetful. Namely, the adversary may learn the internal state such as the private key of a corrupted node, but if the node recovers, then the adversary must forget the private key. This is realistic because otherwise the adversary can eventually learn all the private keys and sabotage the system.

C. Extended TrustChain

The primary data structure used in our system is the Extended TrustChain. Each node u has a public and private key pair—pk_u and sk_u, and a chain B_u. The chain consist of blocks $B_u = \{b_{u,i} : i \in \{0, \dots, h-1\}\}$, where $b_{u,i}$ is the ith block of u, and h is the height of the block (i.e. $h = |B_u|$). We often use $b_{u,h}$ to denote the latest block. There are two types of blocks, TX blocks and CP blocks. If T_u is the set of all TX blocks in B_u and C_u is the set of all CP blocks is B_u, then it must be the case that $T_u \cup C_u = B_u$ and $T_u \cap C_u = \varnothing$. The notation $b_{u,i}$ is generic over the block type. We assume there exists a function $\mathsf{typeof} : B_u \to \{\tau, \gamma\}$ that returns the type of the block, where τ represents the TX type and γ represents the CP type.

Definition 2. Transaction block

The TX block is a six-tuple, i.e

$$t_{u,i} = \langle \mathsf{H}(b_{u,i-1}), seq_u, txid, pk_v, m, sig_u \rangle.$$

We describe each item in turn.

1) $\mathsf{H}(b_{u,i-1})$ *is the hash pointer to the previous block.*
2) seq_u *is the sequence number which should equal* i.
3) $txid$ *is the transaction identifier, it should be generated using a cryptographically secure pseudo-random number generator by the initiator of the transaction.*
4) pk_v *is the public key of the counterparty* v.
5) m *is the transaction message.*
6) sig_u *is the signature created using* sk_u *on the concatenation of the binary representation of the five items above.*

The fact that we have no constraint on the content of m *is in alignment with our design goal—application neutrality.*

TX blocks come in pairs. In particular, for every block

$$t_{u,i} = \langle \mathsf{H}(b_{u,i-1}), seq_u, txid, pk_v, m, sig_u \rangle$$

there exist one and only one pair

$$t_{v,j} = \langle \mathsf{H}(b_{v,j-1}), seq_v, txid, pk_u, m, sig_v \rangle.$$

Note that the $txid$ *and* m *are the same, and the public keys refer to each other. Thus, given a TX block, these properties allow us to identify its pair.*

Definition 3. Checkpoint block

The CP block is a five-tuple, i.e.

$$c_{u,i} = \langle \mathsf{H}(b_{u,i-1}), seq_u, \mathsf{H}(\mathcal{C}_r), r, sig_u \rangle,$$

where $\mathcal{C}_r$ *is the consensus result (which we describe in Definition 4) in round* r*, the other items are the same as the TX block definition.*

The genesis block in the chain must be a CP block in the form of

$$c_{u,0} = \langle \mathsf{H}(\perp), 0, \mathsf{H}(\perp), 0, sig_u \rangle,$$

where $\mathsf{H}(\perp)$ *can be interpreted as applying the hash function on an empty string. The genesis block is unique because every node has a unique public and private key pair.*

Definition 4. Consensus result

Our consensus protocol runs in rounds. Every round is identified by a round number r*, which is incremented on every new round. The consensus result is a tuple, i.e.* $\mathcal{C}_r = \langle r, C \rangle$*, where* C *is a set of CP blocks agreed by the facilitators of round* r.

Here we define an important property which results from the interleaving nature of CP and TX blocks. It is used in our validation protocol.

Definition 5. Enclosure and agreed enclosure

If there exist a tuple $\langle c_{u,a}, c_{u,b} \rangle$ *for a TX block* $t_{u,i}$*, where*

$$a = \operatorname*{arg\,min}_{k, k<i, typeof(b_{u,k})=\gamma} (i - k)$$

$$b = \operatorname*{arg\,min}_{k, k>i, typeof(b_{u,k})=\gamma} (k - i),$$

then $\langle c_{u,a}, c_{u,b} \rangle$ *is the enclosure of* $t_{u,i}$*. Intuitively, that is the two closest CP blocks that surround* $t_{u,i}$*. Note that* $c_{u,a}$ *is the more recent CP block. Also, some TX blocks may not have any enclosure, then its enclosure is* $\perp$*. Agreed enclosure is the same as enclosure with an extra constraint where the CP blocks must be in some consensus result* $\mathcal{C}_r$.

Definition 6. Fragment and agreed fragment

If the enclosure of some TX block $t_{u,i}$ *is* $\langle c_{u,a}, c_{u,b} \rangle$*, then its* fragment $F_{u,i}$ *is defined as* $\{b_{u,i} : a \leq i \leq b\}$.

[2] Reliability can be achieved in unreliable networks by resending messages or using some error correction code.

Similarly, agreed fragment *has the same definition as fragment but using agreed enclosure. For convenience, the function* $\mathsf{agreed\_fragment}(\cdot)$ *represents the agreed fragment of some TX block if it exists, otherwise* $\perp$.

D. Consensus Protocol

Our scalable consensus protocol runs on top of Extended TrustChain. It uses an unmodified asynchronous subset consensus (ACS) algorithm as the key building block. The objectives of the protocol are to allow honest nodes always make progress (in the form of creating new CP blocks), compute correct consensus result in every round and have an unbiased election of facilitators. Concretely, we define the necessary properties as follows.

Definition 7. Properties of the consensus protocol $\forall r \in \mathbb{N}$, *the following properties must hold.*

1) Agreement*: If one correct node outputs a set of facilitators* $\mathcal{F}_r$*, then every node outputs* $\mathcal{F}_r$
2) Validity*: If any correct node outputs* $\mathcal{F}_r$*, then*
   a) $|\mathcal{C}_r| \geq N - t$ *and each CP in* $\mathcal{C}_r$ *belong to a different node*[3]*.*
   b) $\mathcal{F}_r$ *must contain at least* $n - t$ *honest nodes and*
   c) $|\mathcal{F}_r| = n$*.*
3) Fairness*: Every node with a CP block in* $\mathcal{C}_r$ *should have an equal probability of becoming a member of* $\mathcal{F}_r$*.*
4) Termination*: Every correct node eventually outputs some* $\mathcal{F}_r$*.*

These properties are similar to Byzantine consensus properties but there are subtle differences. Firstly, they are properties for every node in the network and not just the facilitators. Secondly, they must be satisfied for all rounds because the whole system falls apart if one of the property cannot be satisfied in one of the rounds.

1) Bootstrap phase: As with many distributed systems, there must be a bootstrap phase which sets up the system. Imagine that there is some bootstrap oracle, that initiates the code on every node. The code satisfies all the properties in Definition 7. Namely, every node has the same set of valid facilitators $\mathcal{F}_1$ that are randomly chosen. This concludes the bootstrap phase. For any future rounds, the consensus phase is used.

In practice, the bootstrap oracle is most likely the software developer and some of the desired properties cannot be achieved. In particular, it is not possible to have the fairness property because it is unlikely that the developer knows the identity of every node in advance.

2) Consensus phase: The consensus phase begins when $\mathcal{F}_r$ is available to all the nodes. Note that $\mathcal{F}_r$ indicates the facilitators that were elected using results of round r and are responsible for driving the ACS protocol in round $r + 1$. The goal is to reach agreement on a set of new facilitators $\mathcal{F}_{r+1}$ that satisfies the four properties in Definition 7.

There are two scenarios in the consensus phase. First, if node u is not the facilitator, it sends $\langle \mathtt{cp\_msg}, c_{u,h} \rangle$ to all the facilitators. Second, if the node is a facilitator, it waits until $N - t$ messages of type cp_msg are received. Invalid messages are removed. That is blocks with invalid signatures and blocks signed by the same key. With the sufficient number of cp_msg messages, it begins the ACS algorithm and some $\mathcal{C}'_{r+1}$ should be agreed upon by the end of it. Duplicates and blocks with invalid signatures are again removed from $\mathcal{C}'_{r+1}$ and we call the final result $\mathcal{C}_{r+1}$. We have to remove invalid blocks a second time (after ACS) because the adversary may send different CP blocks to different facilitators, which results in invalid blocks in the ACS output, but not in any of the inputs.

The core of the consensus phase is the ACS protocol. While any ACS protocol that satisfies the standard definition will work, we use a simplification of HoneyBadgerBFT [5] as our ACS protocol because it is a consensus algorithm designed for blockchain systems. We do not use the full HoneyBadgerBFT due to the following. First, the transactions in HoneyBadgerBFT are first queued in a buffer and the main consensus algorithm starts only when the buffer reaches an optimal size. We do not have an infinite stream of CP blocks, thus buffering is unsuitable. Second, HoneyBadgerBFT uses threshold encryption to hide the content of the transactions. But we do not reach consensus on transactions, only CP blocks, so hiding CP blocks is meaningless for us as it contains no transaction information.

Continuing, when $\mathcal{F}_r$ reaches agreement on $\mathcal{C}_{r+1}$, they immediately broadcast[4] two messages to all the nodes—first the consensus message $\langle \mathtt{cons\_msg}, \mathcal{C}_{r+1} \rangle$, and second the signature message $\langle \mathtt{cons\_sig}, r, sig \rangle$. The reason for sending cons_sig is the following. Recall that channels are not authenticated, and there are no signatures in $\mathcal{C}_{r+1}$. If a non-facilitator sees some $\mathcal{C}_{r+1}$, it cannot immediately trust it because it may have been forged. Thus, To guarantee authenticity, every facilitator sends an additional message that is the signature of $\mathcal{C}_{r+1}$.

Upon receiving $\mathcal{C}_{r+1}$ and at least $n - f$ valid signatures by some node u, u performs two tasks. First, it creates a new CP block using $\mathsf{new\_cp}(\mathcal{C}_{r+1}, r + 1)$ using Algorithm 1. Second, it computes the new facilitators using $\mathsf{get\_facilitator}(\mathcal{C}_{r+1}, n)$ using Algorithm 2, and updates its facilitator set to the result. This concludes the consensus phase and brings us back to the beginning of the consensus phase.

Algorithm 1 Function $\mathsf{new\_cp}(\mathcal{C}_r, r)$ runs in the context of the caller u. It creates a new CP block and appends it to u's chain.

```
$h \leftarrow |B_u|$
$c_{u,h} \leftarrow \langle \mathsf{H}(b_{u,h-1}), h, \mathsf{H}(\mathcal{C}_r), r, sig_u \rangle$
$B_u \leftarrow B_u \cup c_{u,h}$
```

[3] $\mathcal{C}_r$ is a tuple but we abuse the notation here by writing $|\mathcal{C}_r|$ to mean the number of CP blocks in the second element of $\mathcal{C}_r$.

[4] An alternative to using broadcasting is to use an epidemic protocl [13], as long as it can guarantee delivery.

Algorithm 2 Function get_facilitator($\mathcal{C}_r, n$) takes the consensus result $\mathcal{C}_r$ and an integer n, then sorts the CP blocks C by the luck value (the λ-expression), and outputs the smallest n elements.

```
$\langle r, C\rangle \leftarrow \mathcal{C}_r$
take($n$, sort_by($\lambda x.\mathsf{H}(\mathcal{C}_r || pk$ of $x$), $C$))
```

Our protocol has some similarities with synchronizers [14, Chapter 10] because it is effectively a technique to introduce synchrony in an asynchronous environment. If we consider the facilitators as a collective authority, then it can be seen as a synchronizer that sends pulse messages (in the form of `cons_msg` and `cons_sig`) to indicate the start of a new clock pulse. Every node then sends a completion messages (in the form of `cp_msg`) to the new collective authority to indicate that they are ready for the next pulse.

E. Transaction Protocol

The TX protocol, shown in Algorithm 4, is run by all nodes. It is also known as True Halves, first described by Veldhuisen [15, Chapter 3.2]. Nodes that wish to initiate a transaction calls new_tx($pk_v, m, txid$) (Algorithm 3) with the intended counterparty v identified by pk_v and message m. $txid$ should be a uniformly distributed random value, i.e. $txid \in_R \{0,1\}^{256}$. Then the initiator sends $\langle \texttt{tx\_req}, t_{u,h}\rangle$ to v.

Algorithm 3 Function new_tx($pk_v, m, txid$) generates a new TX block and appends it to the caller u's chain. It is executed in the private context of u, i.e. it has access to the sk_u and B_u. The necessary arguments are the public key of the counterparty pk_v, the transaction message m and the transaction identifier $txid$.

```
$h \leftarrow |B_u|$
$t_{u,h} \leftarrow \langle \mathsf{H}(b_{u,h-1}), h, txid, pk_v, m, sig_u\rangle$
$B_u \leftarrow B_u \cup \{t_{u,h}\}$
```

Algorithm 4 The TX protocol which runs in the context of node u.

```
Upon $\langle \texttt{tx\_req}, t_{v,j}\rangle$ from $v$
    $\langle \_, \_, txid, pk_v, m, \_\rangle \leftarrow t_{v,j}$
    new_tx($pk_u, m, txid$)
    store $t_{v,j}$ as the pair of $t_{u,h}$
    send $\langle \texttt{tx\_resp}, t_{u,h}\rangle$ to $v$
Upon $\langle \texttt{tx\_resp}, t_{v,j}\rangle$ from $v$
    $\langle \_, \_, txid, pk_v, m, \_\rangle \leftarrow t_{v,j}$
    store $t_{v,j}$ as the pair of the TX with identifier $txid$
```

A key feature of the TX protocol is that it is non-blocking. At no time in Algorithm 3 or Algorithm 4 do we need to hold the chain state and wait for some message to be delivered before committing a new block to the chain. This allows for high concurrency where we can call new_tx($\cdot$) multiple times without waiting for the corresponding `tx_resp` messages.

F. Validation protocol

Up to this point, we do not provide a mechanism to detect tampering. The validation protocol aims to solve this issue. The protocol is also a request-response protocol, just like the transaction protocol. But before explaining the protocol itself, we first define what it means for a transaction to be valid.

1) Validity definition: A transaction can be in one of three states in terms of validity—*valid*, *invalid* and *unknown*. Given a fragment $F_{v,j}$, the validity of the TX block $t_{u,i}$ and the agreed fragment of it is captured by the function get_validity($t_{u,i}, F_{u,i}, F_{v,j}$) (Algorithm 5). Note that $t_{u,i}$ and $F_{u,i}$ are assumed to be valid, otherwise the node calling the function would have no point of reference. This is not difficult to achieve. Typically the caller is u, so it knows its own TX block and the corresponding agreed fragment. If the caller is not u, it can always query for the agreed fragment that contains the transaction of interest from u.

Algorithm 5 works as follows. Before Algorithm 5, we essentially check whether the fragment is the one that the verifier needs. If it is not, then the verifier cannot make any decision and return *unknown*. This is likely to be the case for new transactions because the result of agreed_fragment($\cdot$) would be $\bot$. The next two conditions check for tampering or missing blocks, if any of these misconducts are detected, then the TX is invalid.

We stress that the *unknown* state means that the verifier does not have enough information to continue with the validation protocol. If enough information is available at a later time, then the verifier will output either *valid* or *invalid*.

Note that the validity is on a transaction (two TX blocks with the same $txid$), rather than on one TX block owned by a single party. It is defined this way because the malicious sender may create new TX blocks in their own chain but never send `tx_req` messages. In that case, it may seem that the counterparty, who is honest, purposefully omitted TX blocks. But in reality, it was the malicious sender who did not follow the protocol. Thus, in such cases, the whole transaction identified by its $txid$ is marked as invalid.

Further, the caller of get_validity($t_{u,i}, F_{u,i}, F_{v,j}$) is not necessarily u[5] Any node w may call get_validity($t_u u, i, F_{u,i}, F_{v,j}$) as long as w has an the necessary input parameters. $F_{u,i}$ and $t_{u,i}$ may be readily available if $w = u$. Otherwise, w could query u with the `vd_req` message and then obtain the result from `vd_resp`. This is the validation protocol which we describe next.

2) Validation protocol: Our validation protocol, shown in Algorithm 6, is designed to classify transactions according to the aforementioned validity definition. If u wishes to validate some TX with ID $txid$ and counterparty v, it sends $\langle \texttt{vd\_req}, txid\rangle$ to v. The desired properties of the validation protocol are as follows.

Definition 8. Properties of the validation protocol

[5] In practice, it often is because after completing the TX protocol the parties are incentivised to check that the counterparty "did the right thing".

Algorithm 5 Function get_validity$(t_{u,i}, F_{u,i}, F_{v,j})$ validates the transaction represented by $t_{u,i}$. We assume $F_{u,i}$ is always correct and contains $t_{u,i}$. $F_{v,j}$ is the corresponding fragment received from v.

```
if $F_{v,j}$ is not a fragment created in the same round as $F_{u,i}$ then
    return unknown
$\langle \_, \_, txid, pk_v, m, \_ \rangle \leftarrow t_{u,i}$
if number of blocks of $txid$ in $F_{v,j} \neq 1$ then
    return invalid
$t_{v,j} \leftarrow$ the TX block with $txid$ in $F_{v,j}$
$\langle \_, \_, txid', pk_u, m', \_ \rangle \leftarrow t_{v,j}$
if $m \neq m' \vee txid \neq txid'$ then
    return invalid
if $t_{u,i}$ is not signed by $pk_u \vee t_{v,j}$ is not signed by $pk_v$ then
    return invalid
return valid
```

1) Agreement*: If any correct node decides on the validity (except when it is* unknown*) of a transaction, then all other correct nodes are able to reach the same conclusion or* unknown.
2) Validity*: The validation protocol outputs the correct result according to the aforementioned validity definition.*
3) Liveness*: Any valid (invalid) transaction is marked as valid (invalid) eventually.*

Algorithm 6 Validation protocol which runs in the context of u

```
Upon $\langle$vd_req$, txid\rangle$ from $v$
    $t_{u,i} \leftarrow$ the transaction identified by $txid$
    $F_{u,i} \leftarrow$ agreed_fragment$(t_{u,i})$
    send $\langle$vd_resp$, txid, F_{u,i}\rangle$ to $v$
Upon $\langle$vd_resp$, txid, F_{v,j}\rangle$ from $v$
    $t_{u,i} \leftarrow$ the transaction identified by $txid$
    if $F_{u,i}$ and $F_{v,j}$ are available and $F_{u,i}$ is the agreed fragment of $t_{u,i}$ then
        set the validity of $t_{u,i}$ to get_validity$(t_{u,i}, F_{u,j}, F_{v,j})$
```

We make two remarks. First, just like the TX protocol, we do not block at any part of the protocol. Second, suppose some $F_{v,j}$ validates $t_{u,i}$, then that does not imply that $t_{u,i}$ only has one pair $t_{v,j}$. Our validity requirement only requires that there is only one $t_{v,j}$ in the correct consensus round. The counterparty may create any number of fake pairs in later consensus rounds. But these fake pairs only pollutes the chain of v and can never be validated because the round is incorrect.

G. Design variations and tradeoffs

Up to this point, we have discussed our protocol in the context of the model and assumptions defined in Section III-B. In this section, we explore a few design variations which we can make, some of them require a relaxed version of our original model. They enable better performance, allow us to apply our design in the fully permissionless setting and improves privacy.

1) Using timing assumption in the permissionless setting: Our model is purely asynchronous, where we make no timing assumptions anywhere in the protocol. However, in many applications it is often fine to make timing assumptions. For example, TCP relies on timeout for its retransmission and the `nLockTime` property in Bitcoin transactions makes the transaction unspendable until some time in the future (either Unix time or block height) [16]. One limitation of our system is that we use the parameter N in our algorithms, which makes it unsuitable for the permissionless environment where users can join and leave at will. In this section we show how making a timing assumption would allow us to operate in the permissionless setting.

At the start of our consensus phase (Section III-D2), facilitators must wait for $N - f$ `cp_msg` messages. This is the only place where we used N as a parameter. To introduce timing, instead of waiting for $N - f$ messages, we wait for some time D, such that D is sufficiently long for honest nodes to send their CP blocks to the facilitators. Consequently, this removes the need for a PKI because the collected CP blocks may be from nodes that nobody has seen in the past. However, choosing the parameter D is difficult and depends on a number of factors such as the network condition, message size, and so on. Overestimating it would make agreed fragments much longer than usual, which increases communication costs for validation. Underestimating it would lead to unfairness where users with a poor internet connection will have a lower chance to be selected as a facilitator in the next round. Nevertheless, there is a significant gain for making the timing assumption and that is the ability to operate in the permissionless setting which we explain next.

Suppose a new node wish to join the network and the facilitators are known (this can be done with a public registry). It simply sends its latest CP block to the facilitators. Then, in the next round, the node will have a chance to become a facilitator just like any existing node. To leave the network, nodes simply stop submitting CP blocks. There is a subtlety here which happens when the node is elected as a facilitator in the following round. In this case, the node must fulfil its obligation by completing the consensus protocol, but without proposing its own CP block, before leaving. Otherwise, the $n \geq 3t + 1$ condition may be violated.

2) Privacy preserving validation protocol using compact blocks: Our approach already has privacy preserving features in comparison to early blockchain systems. That is, the transactions for each node are only revealed during the validation protocol. Hence if two nodes never directly or indirectly interact with each other, their transactions are never revealed.

We can take our privacy-preserving property one step further by introducing another level of hash pointer indirection. The result is shown in Figure 2.

Concretely, we introduce an additional block type, namely compact block. Such blocks only have three attributes,

1) Seq—the sequence number its corresponding block,

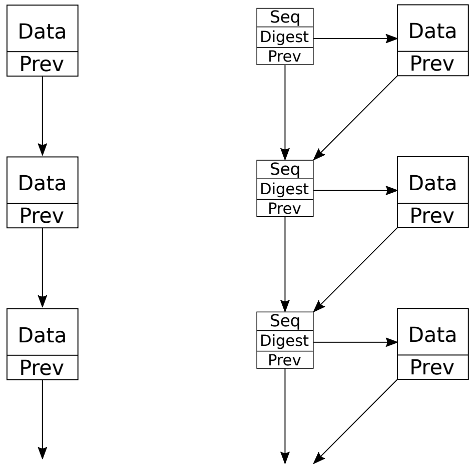

Fig. 2: The chain on the left represent direct chaining, where the digest in "Prev" is simply the digest of the previous block. The chain on the right uses compact blocks, represented by the smaller squares. Compact blocks also form a chain as before, but they each have a hash pointer to the full block, identified by "Seq" and "Digest".

2) Digest—the digest of its corresponding block, and
3) Prev—the digest of the previous compact block.

Each compact block has a corresponding full block (either a CP block or a TX block). The relationship is uniquely identified with Seq or Digest. Recall that our original validation protocol requires the nodes to send the full agreed fragment. With compact blocks, it is only necessary to send the compact version of the agreed fragment. The validation then proceeds in a similar fashion, provided that the pair of the to-be-validated TX block is known.

3) Optimising validation protocol using cached agreed fragments: One more way to improve the efficiency of the validation protocol is to use a single agreed fragment to validate multiple transactions. Concretely, upon receiving an agreed fragment from node A, rather than validating a single transaction, we attempt to validate all transactions with A in the unknown state in that fragment.

For a node, the benefit of this technique is maximised when it only transacts with one other node. In this case, the communication of one fragment is sufficient to validate all transactions in that fragment. In the opposite extreme, if every transaction that the node makes is with another unique node, then the caching mechanism would have no effect.

4) Global fork detection: The validation algorithm guarantees that there are no forks within a single agreed fragment. This is sufficient for some applications such as proving the existence of some information. However, for applications such as digital cash where every block depends on one or more previous blocks, our scheme is not suitable. For such applications, we need to guarantee that there are no forks from the genesis block leading up to the TX block of interest.

There are a variety of approaches to do global fork detection. First and the easiest solution is to simply ask for the complete chain of the counterparty. The verifier can be sure that there are no forks if the following holds.

1) The hash pointers are correct.
2) All the CP blocks are in consensus.
3) The TX of interest is in the chian.

We use this approach in our prior work on Implicit Consensus [17]. It may sound inefficient at first, but nodes employ caching to minimise communication costs.

The second approach is probabilistic but with only a constant communication overhead over our current design. For a node, observe that if all of its agreed fragments has a transaction with an honest node, then the complete chain is effectively validated in a distributed manner. The only way for an attacker to make a fork is to make sure that the agreed fragment containing the fork has no transactions with honest nodes. Such malicious behaviour is prevented probabilistically using a challenge-response protocol as follows. Suppose node A wish to make a transaction with node B. A first sends a challenge to B asking it to prove that it holds a valid agreed fragment between some consensus round specified by A. If B provides a correct proof, then they run the transaction protocol as usual. If B provides an invalid proof or refuses to respond, then A would refuse to make the transaction.

IV. Correctness and fault tolerance analysis

Here we evaluate our system analytically to ensure the desired properties (Definition 7 and Definition 8) hold. An informal argument is given in this section, we refer to **checo**?? for an indepth analysis.

A. Correctness of the consensus protocol

Recall that the consensus protocol properties (Definition 7) has four conditions—agreement, validity, fairness and termination. Agreement, validity and termination hold due to the fact that the following three facts

1) The CP blocks sent to the facilitators are eventually delivered and then ACS eventually starts.
2) Agreement, validity and termination holds for ACS.
3) The consensus result and signatures are eventually disseminated to all the nodes.

Fairnress holds if we model $\mathsf{H}(\cdot)$ as a query to the random oracle. Sorting nodes by the luck value $\mathsf{H}(\mathcal{C}||pk)$ can be seen as a random permutation of a list of nodes. Therefore every node has the same probability of becoming a facilitator.

B. Correctness of the validation protocol

Using the previous result, we show that the agreement and validity properties (from Definition 8) hold for the validation protocol.

The validty property holds because we use $\mathsf{get\_validity}(\cdot)$ in the validation protocol. The agreement property holds due to the collision resistent property of cryptographically secure hash function. Suppose two honest nodes decided on two different state, *valid* and *invalid* for the same transaction. Due to the properties of the consensus protocol, the agreed enclosure for the transaction must be the same. For that to happen, two agreed fragments must exist for the same transaction. Recall that blocks form a hash chain. So this cannot happen unless the hash function is not cryptographically secure.

Liveness, unfortunately does not hold in our model. A malicious node can ask honestly when running the transaction protocol, but then never respond to any validation requests. Therefore there are transactions that can never be validated. Nevertheless, the malicious node will be at an economical loss if it is not responsive because honest nodes are less likely to make contact with nodes that do not respond to validation messages.

A stronger version of the validity definition exists. That is, if two honest nodes make a transaction, then the transaction state is always valid in addition to our current validity definition. Under our purely asynchronous model, we cannot guarantee this stronger version. Since the adversary can delay any message for any amount of time, it can make sure all `tx_req` messages are delivered in a round later than the round which the message is sent. Effectively, the transaction pair would always be in different rounds and the validation protocol would not output *valid*. We believe in a relaxed model, i.e. a weakly synchronous model, a stronger validity definition is possible.

C. Fault tolerance

Finally, we consider the effect when the number of adversaries is more than t. This is useful because in practice it is difficult to guarantee that t satisfies $n \geq 3t+1$, especially when N is large. Hence we are interested in the probability for this to happen under our facilitator election process. The problem is modeled by a hypergeometric distribution, where the random variable X is the number of malicious nodes. For our system to fail, we are interested in the following probability.

$$\Pr[X \geq \lfloor \frac{n-1}{3} \rfloor + 1]$$

Let $E[X] = n\alpha$ where $0 \leq \alpha \leq \lfloor \frac{n-1}{3} \rfloor / n < (\lfloor \frac{n-1}{3} \rfloor + 1)/n$. Using the tail inequality found in [18]

$$\Pr[X \geq E[X] + \tau n] \leq e^{-2\tau^2 n},$$

and setting

$$\tau = \frac{\lfloor \frac{n-1}{3} \rfloor + 1}{n} - \alpha,$$

we arrive at the following bound

$$\Pr[X \geq \lfloor \frac{n-1}{3} \rfloor + 1] \leq e^{-2\left(\frac{\lfloor \frac{n-1}{3} \rfloor + 1}{n} - \alpha\right)^2 n}.$$

The bound is not tight, but it is useful for picking parameters. If n is known, then we can pick a α such that the probability becomes small. On the other hand, if α is fixed, but small, we may increase n to achieve the same. Furthermore, hypergeometric distributions have light tails with "faster-than-exponential fall-off" [18], the probability for picking more than $\lfloor \frac{n-1}{3} \rfloor$ malicious nodes drops of exponentially as the expected value moves away from $\lfloor \frac{n-1}{3} \rfloor$. To put this into perspective, suppose $n = 1000$ and $\alpha = 1/5$, i.e. a fifth of the nodes are malicious. Then the probability to draw more black balls than the threshold is only 2.6×10^{-16}.

V. Implementation and Evaluation

The prototype implementation can be found on GitHub.

https://github.com/kc1212/consensus-thesis-code

It implements the three protocols and the Extended TrustChain. We also implement two optimisations—privacy preserving validation protocol using compact blocks (Section III-G2) and optimised validation protocol using cached agreed fragments (Section III-G3). It is written in the event driven paradigm, using the Python programming language[6]. The cryptography primitives we use are SHA256 for hash functions and Ed25519 for digital signatures. Both of which are provided by libnacl [7].

A. Experimental setup

We run the experiment on the DAS-5[8] with up to 1200 nodes. Every node makes a transaction with a random node twice every second. Since Bitcoin transactions are approximately 500 bytes [19], we use a uniformly random transaction size sampled between 400 and 600 bytes.

To coordinate nodes on many different machines, we employ a discovery server to inform every node the IP addresses and port numbers of every other node. It is only run before the experiment and is not used during the experiment.

B. Linear Global Throughput

The global throughput results are shown in Figure 3. Evidently, the throughput has a linear relationship with the population size. This result is a strong indication of the horizontal scalability which we aimed to achieve.

VI. Conclusion

In this work we introduced an application neutral blockchain system which we call Checo. We add checkpoint block to the existing TrustChain data structure for use in our consensus protocol. The round based consensus protocol uses ACS as a building block to reach consensus on checkpoint blocks. The consensus result, which is a set of checkpoint blocks, lets nodes elect new facilitators and create new checkpoint blocks. To make transactions, nodes use the transaction protocol, which is adapted from an existing work called True Halves. Finally, we introduce a validation protocol which ensures that if agreed fragments for some transaction exists, then nodes reach agreement on the validity of that transaction. The novelty of the validation protocol is that it uses point-to-point communication, i.e. nodes only validate the transactions of interest, this enables our horizontal scalability property.

[6] https://www.python.org/

[7] https://pypi.python.org/pypi/libnacl

[8] https://www.cs.vu.nl/das5/

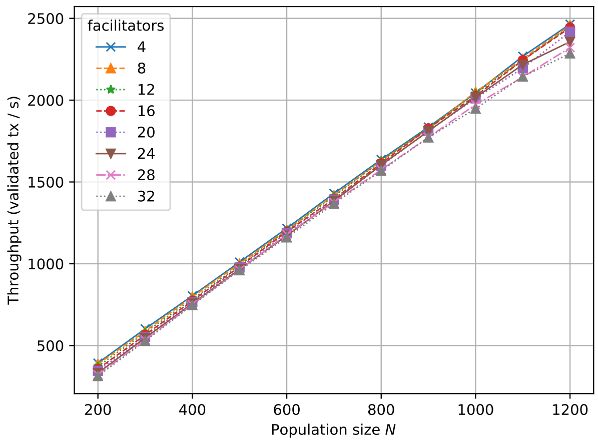

Fig. 3: Global throughput increases as the population increases when every node transact at the same rate. Using fixed neighbours results in a higher throughput because of the caching mechanism.

References

[1] J. Poon and T. Dryja, "The bitcoin lightning network," Jan. 2016. [Online]. Available: https://lightning.network/lightning-network-paper.pdf.

[2] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*, Springer, 2015, pp. 3–18.

[3] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[4] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[5] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 31–42.

[6] L. Luu, V. Narayanan, K. Baweja, C. Zheng, S. Gilbert, and P. Saxena, "Scp: A computationally-scalable byzantine consensus protocol for blockchains.," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1168, 2015.

[7] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016, pp. 279–296.

[8] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus," *ArXiv preprint arXiv:1612.02916*, 2016.

[9] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[10] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 17–30.

[11] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger.," *IACR Cryptology ePrint Archive*, vol. 2017, p. 406, 2017.

[12] S. Norberhuis, *Multichain: A cybercurrency for cooperation*, Dec. 2015. [Online]. Available: http://resolver.tudelft.nl/uuid:59723e98-ae48-4fac-b258-2df99d11012c.

[13] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[14] R. Wattenhofer, *Principles of distributed computing*, 2016. [Online]. Available: http://dcg.ethz.ch/lectures/podc_allstars/lecture/podc.pdf.

[15] P. Veldhuisen, "Leveraging blockchains to establish cooperation," Master's thesis, Delft University of Technology, May 2017. [Online]. Available: http://resolver.tudelft.nl/uuid:0bd2fbdf-bdde-4c6f-8a96-c42077bb2d49.

[16] Bitcoin Project. (). Bitcoin developer guide, [Online]. Available: https://bitcoin.org/en/developer-guide (visited on 07/04/2017).

[17] Z. Ren, K. Cong, J. Pouwelse, and Z. Erkin, *Implicit consensus: Blockchain with unbounded throughput*, 2017. eprint: arXiv:1705.11046.

[18] M. Skala, "Hypergeometric tail inequalities: ending the insanity," *ArXiv e-prints*, Nov. 2013. arXiv: 1311.5939 `[math.PR]`.

[19] TradeBlock. (Oct. 2015). Analysis of bitcoin transaction size trends, [Online]. Available: https://tradeblock.com/blog/analysis-of-bitcoin-transaction-size-trends (visited on 07/14/2017).