CantonCoin: Gaining Horizontal Scalability and Privacy with Distributed Commits Instead of Global Consensus

Andreas Lochbihler*, Ognjen Marić*, and Derek H. Sorensen†

*Digital Asset (Switzerland) GmbH, Zurich, Switzerland
andreas.lochbihler,ognjen.maric@digitalasset.com

†Digital Asset, New York City, New York, USA derekhsorensen@gmail.com

Abstract

Bitcoin sought to create a trustless and censorship-resistant payment system. However, to obtain a practical day-to-day payment system capable of serving the global needs, these goals must be balanced with many other properties, such as scalability and privacy. In this paper, we show that replacing state machine replication (the mechanism underlying Bitcoin and most other blockchains) with a distributed commit protocol over private local ledgers removes the scalability bottlenecks and enhances privacy. We demonstrate our approach using the Canton commit protocol, resulting in the *CantonCoin* prototype cryptocurrency.

1 Introduction

The cryptocurrency wave started with Bitcoin [23], whose goal was to enable payments that are trustless (require no mutual trust or trusted third parties) and permissionless (open to any two parties). Bitcoin's model uses state machine replication backed by a proof-of-work consensus mechanism to ensure the consistency of a global ledger among an open set of participants. Bitcoin owners are pseudonymous, identified only by their addresses, i.e., public keys. To date, Bitcoin's approach is arguably the most successful one in achieving its stated goals, although this success is undermined by the proof of work economics that lead to centralization [11]. Despite this success, Bitcoin has not supplanted the traditional financial system for day-to-day use, and its average number of daily transaction is stagnating in 2019. A practical global system must provide additional properties, which must be balanced against the Bitcoin's goals. In this paper, we focus on two such properties.

1. **Scalability** Perhaps the best-known limitation of Bitcoin is the lack of scaling. Bitcoin can process about 7 transactions per second, a figure that is similar for other systems based on proof of work (Ethereum processes 15). For comparison, Visa's payment network averages 2,500 transactions per second and can process up to 24,000/s [15]. Beyond the consumer market, the New York Stock Exchange processes several billion trades a day.

2. **Privacy** Bitcoin's global ledger is publicly shared with all Bitcoin users and provides full public insight into the transaction graph, i.e., the movement of money between the Bitcoin holders' pseudonyms. The hope is that the pseudonyms provide sufficient anonymity. However, by correlating the ledger data with other information, such as known Bitcoin addresses, IP addresses gathered through the gossip network, and web cookies, it is possible to infer information about and often even completely de-anonymize users [9]. Such privacy issues can pose strategic risks for businesses in, e.g., finance.

These problems inspired many of the "second layer" and "altcoin" solutions. To improve scalability, second-layer solutions move the processing of some of the transactions to a secondary system — e.g, side chains — or to peer-to-peer channels such as in the Lightning Network. Such solutions have their own tradeoffs, for example, the problems of sufficient collateral and finding routes in the Lightning Network [13]. Altcoins usually replace proof of work with other consensus mechanisms. Some mechanisms, such as proof of stake, aim to retain the permissionless nature of the system. Others instead opt for permissioned ledgers. These distribute trust among a fixed set of entities, usually called "validators," and use classic Byzantine fault tolerant (BFT) consensus algorithms, instead of proof-of-work or proof-of-stake. Ripple [25] and Libra [2] use such algorithms. These algorithms achieve much higher throughput, but can be safe only if more than two thirds of validators behave honestly [5]. Arguably, these trust assumptions are not significantly different from the realities of Bitcoin centralization. However, the validators are often also explicitly trusted with the monetary policy, i.e., they can jointly issue or destroy coins in order to, e.g., maintain a stable value.

While an efficient consensus mechanism improves throughput, replicating a state machine (i.e., a global ledger) eventually becomes a scaling bottleneck in altcoins, as each user must process every other user's transactions. The resulting total system load is proportional to the number of transactions multiplied by the number of users. Assuming that every user regularly transacts, the load grows *quadratically* with the number of users. While it is reasonable to assume that designated validators can process tens or hundreds of thousands of transactions per second, this is unrealistic for home users who already struggle with Bitcoin-level loads (as witnessed by Bitcoin SPV clients [11]). This further weakens the trustlessness property and leads to further centralization.

The privacy problem also appears in the other systems with public ledgers, such as Ripple [21], and legislators have voiced concerns over privacy in Libra [24]. The "mixing" solutions such as CoinJoin [8] can help obfuscate the public transaction graph, but are not infallible [12]. Some altcoins such as ZCash [14] and Monero [20] employ zero-knowledge proofs for transaction validation in order to improve privacy and anonymity. These methods hide the transaction amounts and require no public pseudonyms (addresses), at the expense of additional computational complexity. The remaining public information, however, still enables correlation attacks [16, 22].

In this paper we focus on permissioned ledgers, which cryptocurrencies such as Libra and Ripple went for in order to improve scalability. These ledgers give up Bitcoin's original goal of trustless and permissionless payments: they distribute trust, not remove it, and a sufficiently large group of validators can both fake and prevent payments. Yet, their state machine replication approach inherits the privacy and scalability bottleneck issues of a global public ledger. Our main research questions are then: Can a permissioned cryptocurrency by implemented using a primitive different from state machine replication? Can this improve scalability and privacy?

Contributions We show how to implement a permissioned cryptocurrency using distributed commits instead of state machine replication, using the smart contract platform Canton [10]. Instead of a global ledger, Canton maintains a series of private local ledgers, one for each validator and system user, and synchronizes them on demand using its BFT distributed commit protocol. The protocol keeps the local ledgers of non-Byzantine users consistent with each other. We use Canton's open-source smart contract language DAML [3] to prototype a cryptocurrency, and name the resulting system *CantonCoin*. Implementing *CantonCoin* in DAML allows it to be embedded into arbitrary other business processes on Canton. The prototype implementation can be found online [6]. Our approach improves the targeted properties as follows:

- **Scalability** Local ledgers are the key to scalability: a Canton user only processes changes to their ledger. Thus, *CantonCoin* users must process only their own transactions instead of the entirety of world's transactions. Furthermore, Canton's concurrency control mechanisms and sharding via so-called *synchronization domains* enable *CantonCoin* validators to process transactions in parallel. Finally, Canton does not rely on expensive cryptography, such as zero-knowledge proofs or a proof-of-work mechanism. Together, these remove the scalability bottlenecks found in other cryptocurrencies. In particular, as the number of validators is essentially independent of the number of users, the total system load grows *linearly* with the number of users.

- **Privacy** The transaction graph is not publicly accessible, as the local ledgers are private. Consequently, transactions are concealed from non-validators, and the correlation attacks on public ledger are not applicable. Moreover, we design the system such that the money movement is synchronized only among a subset of the validators, not all of them. This gives the users some privacy towards the validators, too, as each validator sees only a fraction of a user's money transactions. Finally, leveraging Canton's *sub-transaction privacy* properties, the system can be extended such that no single validator learns both counterparties to any single transaction.

The rest of the paper is structured as follows. In §2, we describe the main ideas behind *CantonCoin* with an example. We then provide more background on Canton in §3. We describe the *CantonCoin* implementation in more detail in §4, including the background on Canton's open-source smart contract language DAML on the way. In §5, we analyse the properties of *CantonCoin*. We survey the related work in §6, and conclude in §7.

2 *CantonCoin* at a Glance

We consider the following scenario: Alice owns a coin that she wishes to transfer to Bob. We start by describing how Alice can use distributed commits to send her money to Bob such that:

- they do not have to trust each other, and
- no user of *CantonCoin* except for Alice, Bob, and (a subset of) the validators either receives or processes any data about this transfer.

This both removes the scalability bottlenecks and provides privacy. Next, we improve the system's liveness, such that Alice can transfer her coin even if a subset of the validators try to prevent her. Lastly, we provide Alice and Bob with more privacy, by preventing any individual validator from knowing that both Alice and Bob were involved in the transfer.

2.1 Validators and Users

CantonCoin assumes a fixed, universally known set of *validators*, who are jointly trusted with the monetary policy and processing payments. The validators are analogous to those in Libra [2], or the default UNL nodes in Ripple's XRP protocol [7]. Any individual validator may be Byzantine, that is, need not follow the rules prescribed by Canton and *CantonCoin*: it may not send the prescribed messages, or it may send messages different than those prescribed. However, we assume an upper bound on the number of Byzantine operators, which we will denote by f (out of a total of N validators). Non-Byzantine validators are called honest. As usual for permissioned ledgers and BFT algorithms, the parameters f and N are publicly known. For comparison, Libra, assumes $f < N/3$ with $N = 28$, and XRP assumes $f < N/5$ with N being the size of a UNL [7]. We do not fix f now; instead, we will examine the different properties that we can achieve for different choices of f (as a fraction of N).

System's users rely on the validators to receive and transfer coins. We do not assume users to be honest: any number of them can be Byzantine.

2.2 Valid Coins and Transfers

To be deemed to own a valid coin, a user must obtain so-called *endorsements* for the coin from $f + 1$ validators. By the definition of f, any set of $f + 1$ validators contains at least one honest validator: intuitively, we rely on this validator to enforce the correct policies and prevent malicious Alice or Bob from creating money out of thin air. Obviously, we now assume $f < N$.

An endorsement is private data that is shared only between a user and a validator, i.e., replicated at the user and the validator. To transfer a coin from Alice to Bob, the endorsements for Alice's coin must be invalidated and replaced by endorsements for Bob's coin. This must happen transactionally—either all of Alice's endorsements are invalidated (except perhaps those of the Byzantine validators) and all of Bob's are created, or there is no change. Since the endorsements are distributed among multiple entities, we need a *distributed commit* protocol to perform this transaction.

As at least one of Bob's endorsements comes from an honest validator, say V_h, we rely on V_h to ensure that money is not created out of thin air.

The system—consisting of *CantonCoin* smart contract code and the Canton commit protocol—must ensure the following to the honest validator V_h, regardless of what the Byzantine validators and users do:

1. Alice's coin indeed had at least $f + 1$ endorsements, f of which may originate from Byzantine validators.
2. The commit performing the transfer invalidates the old endorsements at all validators who perform the commit correctly. At most f of them might not—but this is OK, as a coin with only f endorsements will not be deemed valid.
3. The commit is atomic, in that the old endorsements will not be visible to any concurrent (or later) commits.

This protects Bob from a malicious Alice, guaranteeing that the money that Bob receives is real and that Alice cannot spend it again. Additionally, to protect Alice from Bob, the system must ensure that the commit only takes place if Alice authorizes it.

Up to now, we have assumed that coins have a unit value. To efficiently solve the case where Alice wants to move a larger amount of coins to Bob, we add an amount to each coin and each endorsement, and allow the splitting and merging of coins.

As the transfer involved only Alice, Bob, and the validators, this simple approach already removes the principal scaling bottlenecks of state machine replication, assuming the commit protocol efficiently handles concurrent commits. It also gives Alice and Bob privacy from the other system users.

2.3 Liveness

While safe for both Alice and Bob, the described system is not live, in the following sense. Ensuring that Alice's coin had $f + 1$ endorsements in the first step of the transfer requires action from their $f + 1$ issuing validators. But a Byzantine validator could refuse to respond and thus prevent Alice from transferring her money to Bob.

To cope, we first assume that Alice can obtain $2f + 1$ endorsements for her coin. Then, if at most f validators refuse to respond (or respond claiming that their endorsement has been invalidated), Alice can abort the attempted commit and attempt another commit with these validators removed. Eventually, she will successfully transfer her coin to Bob, even if any f validators attempt to block her. To be able to transfer his coin later, Bob also wants $2f + 1$ endorsements on his new coin; this requires a total of

$3f + 1$ validators in the system (as f can refuse to issue Bob endorsements). Hence, we now assume $f < N/3$ instead of $f < N$, and assume that the underlying commit protocol can abort commit attempts.

2.4 Sub-transaction Privacy

The sketched system gives Alice and Bob privacy from other system users. Implementing it requires little beyond BFT distributed commits, and such systems have been known since the 1980s [19]. However, most of them show the entirety of the committed transaction to all involved parties. If the protocol provides *sub-transaction privacy*, though, meaning that individual parties see only parts of the transaction that are relevant to them, we can improve the coin's privacy properties further. Namely, we can obtain a system where no single validator learns that Alice sent money to Bob. To that end, we split up our sample transaction into several steps:

1. In the first step, Alice's endorsements are invalidated, and a subset of old endorsers (endorsing validators) of size $f + 1$ issues new endorsements. These endorsements are ownerless, in that the amount's owner is unspecified. This step need be visible only to Alice and her endorsers.

2. Based on these ownerless endorsements, a new set of validators (distinct from Alice's validators) creates a new set of ownerless endorsements. The old ownerless endorsements are deleted. This step must be visible to both sets of endorsers.

3. The new ownerless endorsements are converted into endorsements for Bob. This step need be visible only to Bob and his endorsers.

This way, Alice's endorsers do not learn Bob's identity, and Bob's endorsers do not learn Alice's identity, so that no single validator learns that Alice sent money to Bob. This presumes that Alice's and Bob's endorsers form disjoint sets. Since we already derived the requirement for $2f + 1$ endorsements in each set for liveness, this requires $4f + 2$ validators in total for each transaction. To ensure liveness and be able to tolerate f non-responsive validators, we assume $5f + 2 \leq N$.

We have now seen the basic idea of how mutually distrusting Alice and Bob can transact in a secure, scalable and private way, even with a number of rogue validators in the system. In the remainder, we will show how this idea can be implemented using Canton and its smart contract language DAML, detail the trust assumptions, and analyze the security of the resulting system.

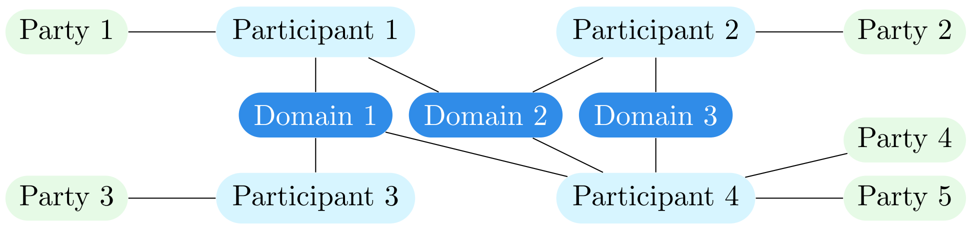

Figure 1: Canton system model

3 Background: Canton in a Nutshell

As established in §2, *CantonCoin* requires a BFT distributed commit protocol, and Canton [10] is precisely such a protocol. In this section, we give a brief overview of the relevant aspects of Canton. We first describe the system model (§3.1), then the underlying data model (§3.2), and finally the commit protocol (§3.3).

3.1 System model

Canton's system model is shown in Figure 1. The system users are called parties. For *CantonCoin*, e.g., they include the validators and the users. Parties run participants that communicate via *synchronization domains*. Canton allows parties to synchronize changes over shared pieces of data. The changes can be made atomic whenever every party runs a participant that is connected to some joint domain. Participants can connect to multiple domains simultaneously.

In the following, we assume that every party runs their own participant and therefore identify the two. Parties can be malicious. Synchronization domains are assumed to be honest-but-curious. They can be run by individual trusted third parties or using BFT replication to distribute trust. Parties must trust the domains they connect to. If a party loses trust in the domain operator(s), they can move their workflows to other domains.

3.2 Data and transaction model

Canton partitions its state space: every piece of data belongs to exactly one partition that is identified by a set of parties. Canton's data objects are called *contracts*, since they are intended to correspond to the parties' rights and obligations. For example, the endorsement that some validator V_1 issues

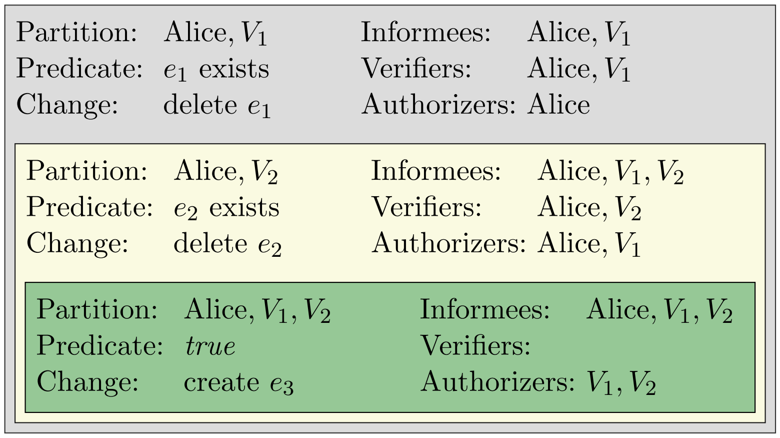

Figure 2: Example action of Alice combining two endorsements e_1 and e_2 into one joint endorsement e_3

for Alice is represented as a contract that belongs to Alice and V_1, i.e., their partition $\{\text{Alice}, V_1\}$.

Data is changed by performing *transactions*, each of which is a list of *actions*. For illustration, we consider a sample action where Alice combines two separate endorsements into a single joint one. More precisely, she combines the endorsements e_1 and e_2 from state partitions $\{\text{Alice}, V_1\}$ and $\{\text{Alice}, V_2\}$ respectively into a single endorsement in the state partition with all three parties $\{\text{Alice}, V_1, V_2\}$. Each action specifies the following parts (Fig. 2 shows them for the sample action in the grey box):

- A single state partition P that the action accesses. In the example, the action to delete e_1 occurs in the partition $P = \{\text{Alice}, V_1\}$.

- A predicate on the state in the partition and a change to its state. Canton contracts are immutable, like in the UTxO model. That is, the state changes can only delete existing contracts and create new ones, but cannot change existing contracts. The sample action's predicate is that e_1 exists, and its change is to delete e_1.

- Its *verifiers*, a set of parties who verify that the predicate holds on the accessed part of the state. This set must be a subset of P. In the example, both Alice and V_1 verify the predicate.

- Its *authorizers*, a set of parties required to authorize the action (Alice in the example).

- Its *informees*, a set of parties who should be notified of the action (Alice and V_1 in the example).
- A *sub-transaction*, i.e., a list of *sub-actions*. This gives the transaction a tree (or more precisely, forest) structure, i.e., transactions are hierarchical. The sub-actions may operate on different partitions. For example, our sample action has a single sub-action (yellow) that deletes another endorsement e_2 from the partition $\{\text{Alice}, V_2\}$. This sub-action has another sub-action of its own (green), which creates a joint endorsement e_3 in the partition $\{\text{Alice}, V_1, V_2\}$, and has no further sub-actions.

Canton's data and transaction model is explained in more detail in [4]. We next explain how transactions are committed.

3.3 Distributed commit protocol

To commit transactions of the form we just described, Canton provides a two-phase commit protocol that is both Byzantine fault tolerant and privacy-preserving. We explain the protocol on the example transaction from Fig. 2 which joins two endorsements. First, Alice encrypts every action (including the sub-actions, their sub-actions etc) of the transaction such that only the action's informees can decrypt it. Alice then creates a message containing this encrypted text and the informees in plaintext. In addition to the informees, it also lists the *confirmers*, which are the union of authorizers and verifiers, in plaintext. These messages are arranged in a Merkle tree according to the original transaction tree structure and sent to a suitable domain in a single batch (Figure 3). In our example, as there are three actions, the batch contains three messages. The first message (grey) goes to Alice and V_1 and the other two messages (yellow and green) go to Alice, V_1, and V_2.

The domain linearly orders and timestamps every batch of messages it receives. It then forwards the encrypted messages, along with the timestamp, to their respective informees. This starts the first (prepare) phase of the two-phase commit protocol.

Every recipient decrypts the messages and checks the correctness of each received message and its action from their perspective. The correctness checks consist of three parts: checking the action's predicates, checking authorization, and checking whether the action itself is allowed and the message is correctly derived. Whether the action is allowed is determined by smart contract code, which is written in DAML. During authorization, the specified authorizers check that they know why the action is happening; for example, Alice can authorize the actions since she requested them herself.

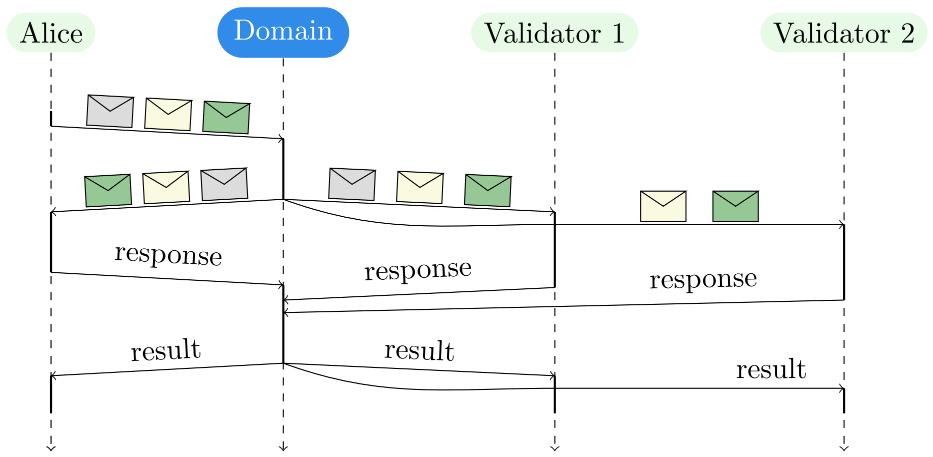

Figure 3: Message flow in Canton

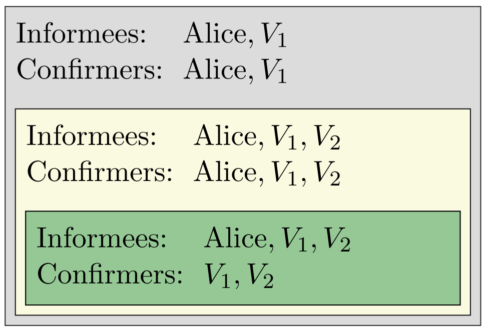
Informees: Alice, V_1
Confirmers: Alice, V_1
Informees: Alice, V_1, V_2
Confirmers: Alice, V_1, V_2
Informees: Alice, V_1, V_2
Confirmers: V_1, V_2

Figure 4: Domain's view on the transaction from Fig. 2

While waiting for the checks to pass, the recipients record the tentative state changes for concurrency control. This way, the checks for unrelated transactions can be performed in parallel. They also respond with the outcome of the checks to the domain. The domain collects and aggregates the responses. If it receives positive responses from all parties that are supposed to respond, it decides that the transaction should be committed. Conversely, if someone sends a rejection, the domain decides to abort the transaction. Responses must be sent within given timeouts, as measured by the message timestamps. Otherwise, the transaction is aborted.

In the second phase of the protocol, the domain sends the decision to all involved participants. After receiving the decision, every participant applies or discards its tentative changes to the local ledger state. The domain is trusted with sending the same decision to everyone, and also with correctly ordering and distributing messages (i.e., implementing atomic multicast). However, Canton includes several mechanisms for detecting domain (or participant) misbehavior: all message exchanges are signed, aborts include the rejection reason such that the submitters can double-check the decision, and participants regularly exchange signatures on their shared state. Privacy-wise, the domain learns the involved participants, so that it can forward the messages appropriately. The transaction data—the state predicates and changes—are encrypted and thus inaccessible to the domain. However, the domain does learn the shape of the transaction, i.e., the number of actions and their nesting, as well as their informees and confirmers. Fig. 4 shows the domain's view on the transaction consisting of the action in Fig. 2.

Recall that participants can connect to multiple domains. The Canton design allows transactions performed over separate domains to be processed completely in parallel. This provides a form of sharding the local ledgers that further improves scalability.

4 *CantonCoin* implementation

CantonCoin is implemented through a series of contracts in Canton's open-source smart contract language, DAML. We now provide details on this implementation, and introduce the relevant DAML concepts on the way (the full language documentation is available online [3]). As a first approximation, DAML is the pure, deterministic fragment of the programming language Haskell (without input/output and other stateful and non-deterministic parts of the language), extended with constructs to define *contract templates*. Canton contracts are instances of the templates, with each instance having a unique identifier. The templates define the data format stored in these instances and the allowed Canton actions on contracts of each template.

4.1 Coins and Endorsements

The two main *CantonCoin* templates are `Coin` and `Endorsement`. A `Coin` contract represents a possibly valid coin. It stores the owner, the amount, and the validators that may endorse it. Its data format is specified in DAML as follows:

```
template Coin with
    owner: Party
    possibleEndorsers: Set Party
    amount: Int
  where
    signatory owner
    observer possibleEndorsers
```

The parties listed under *`signatory`* and *`observer`* together determine the state partition that the contract belongs to. For example, a `Coin` contract with owner Alice and possible endorsers V_1, V_2, V_3 lives in the state partition $\{\text{Alice}, V_1, V_2, V_3\}$. An `Endorsement` contract stores the set of endorsing validators, the contract ID of the endorsed coin, its amount, and the owner. Additionally, an endorsement contract stores the publicly known parameters of the setup: the identities of all validators and the maximum number f of Byzantine validators. The *`ensure`* clause makes sure that all endorsing validators are actually known to be validators.

```
data MarketSetup = MarketSetup with
   allValidators: Set Party
   maxByzantine: Int

template Endorsement with
    validators: Set Party
    coinId: ContractId Coin
    amount: Int
    owner: Party
    marketSetup: MarketSetup
  where
    signatory validators
    observer owner
    ensure validators `subset` marketSetup.allValidators
```

In terms of signatories and observers, endorsements are dual to coins: the validators are the signatories and the coin owner is the observer. This effects how the authorizers and verifiers of actions are defined, and, as we will see

```
template Endorsement
  ...
    choice CombineWith: ContractId Endorsement with
        otherId: ContractId Endorsement
      controller owner
      do
        exercise otherId AddValidators with other = this

    choice AddValidators: ContractId Endorsement with
        other: Endorsement
      controller owner, other.validators
      do
        assert $ (other with validators = validators) == this
        create this with
          validators = other.validators `union` validators
```

Listing 1: Merging Endorsements

in §5, prevents double spends in *CantonCoin* and ensures liveness in the presence of Byzantine validators.

4.2 Merging Endorsements

Having specified the main *CantonCoin* data formats, we now turn to the actions. Every template implicitly specifies a creation action. Its authorizers are the signatories, and its informees are both the signatories and the observers. For, e.g., the `Coin` contract, the owner must authorize the creation, and both the owner and the possible endorsers are the informees. Creation actions do not require anyone to verify any state predicates.

A template may also define choices, which give rise to *exercise actions* in Canton. Intuitively, these allow parties to exercise their rights on the contract. For example, the `Endorsement` template allows the owner to combine its endorsement contract with another one, as shown in §3.2. The corresponding DAML code is shown in Listing 1. Merging endorsements is key to making transfers work with DAML's authorization logic and non-responsive validators.

An exercise action deletes the exercised contract. So, for example, once a `CombineWith` choice has been exercised on an endorsement contract, the contract cannot be exercised on any more, preventing double spends. This action corresponds to the grey action in Fig. 2. The choice's *`controller`* clause specifies the authorizers of the exercise action. Here, the owner must authorize every exercise of the `CombineWith` choice. The informees are the signatories, the observers, and the authorizers. The action's state predicate is that the contract exists. It's verified by all signatories and all authorizers who belong to the contract's state partition. Finally, each **`choice`** specifies its *consequences* in a **`do`** block, which become the sub-actions in the transaction tree. In this example, the `AddValidators` choice on the other endorsement contract is exercised, deleting that contract (corresponds to the yellow action in Fig. 2). The **`assert`** in the **`do`** block of `AddValidators` ensures that the (yellow) action is only possible if the two endorsements differ at most in their validators. As a consequence, this creates the new endorsement contract with the union of the validators (green action).

As mentioned in §3.3, the authorizers of each action check that they know why the action is happening. One such case is when the authorizer has initiated the action herself by submitting the transaction. The other case is when the action is a consequence of an exercise action, and the authorizer is either also an authorizer of this exercise, or a signatory of the contract of this exercise. For example, both V_1 and V_2, who are the authorizers of the endorsement creation, know why it is happening: (1) V_1 is the authorizer (controller) on the `AddValidators` exercise, and (2) V_2 is a signatory of the endorsement contract that `AddValidators` is exercised on. Viewed differently, the DAML code allows V_1 to delegate their control of when to exercise `AddValidators` to the owner, provided that the exercise happens as specified in the `CombineWith` choice.

4.3 Coin transfers

Once endorsements are merged, we can transfer coins. The core logic is implemented using two choices: `Spend` in `Coin` and `Transfer` in `Endorsement`. The `Spend` choice deletes the coin and requires a set of validators as *`controller`*s. These controllers become authorizers on the corresponding exercise action, which, as we will see in §5.1, is key to preventing double spends.

The `Transfer` choice then transfers the endorsement from the owner's coin to a coin of the recipient. The steps are straightforward (code given in Listing 2):

1. Delete the endorsement contract for the owner's coin. This happens implicitly as the choice is exercised on this endorsement contract.

2. Check that the validators on the endorsement are sufficiently many, i.e., form a quorum. This requires the endorsements to have been merged previously.

```
isQuorum: MarketSetup -> Set Party -> Bool
isQuorum m validators = S.size validators > m.maxByzantine

template Coin
  ...
    choice Spend: () with
        validators: Set Party
      controller owner, validators
      do
        assert $ validators `subset` possibleEndorsers

template Endorsement
  ...
    choice Transfer: ContractId Endorsement with
          newOwner: Party
          newCoinId: ContractId Coin
      controller owner
      do
        assert $ isQuorum marketSetup validators
        exercise coinId Spend with validators = this.validators

        newCoin <- fetch newCoinId
        assert $ validators `subset` newCoin.possibleEndorsers
        assert $ newCoin.amount == this.amount
        create this with owner = newOwner, coinId = newCoinId
```
Listing 2: Transferring coins

3. Spend the owner's coin, deleting it and thus invalidating its endorsements.

4. Make sure that the recipient's coin has suitable parameters (same amount and the validators are registered as possible endorsers).

5. Create the endorsement contract for the new coin.

It is now easy to extend this transfer of endorsements into a full coin transfer, visualized in Fig. 5. *CantonCoin* defines a template `TransferRequest` where the recipient specifies the sender, the amount, and the acceptable validators. On such contracts, the sender controls an `AcceptRequest` choice that takes a coin and a list of endorsements as parameter. The consequences of this choice are:

1. A new coin for the recipient is created,

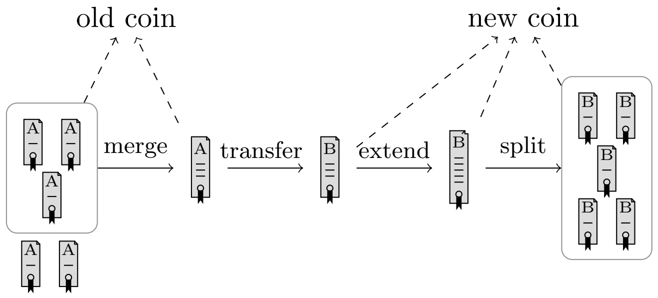

Figure 5: Steps of a coin transfer for $f = 2$

2. The given endorsements are all merged into one endorsement using `CombineWith`,
3. The `Transfer` choice is exercised on the joint endorsement (deleting the old coin),
4. The transferred endorsement is extended to $2f+1$ validators (explained in the next section), and
5. The extended endorsement is split into single-validator endorsements again.

These actions are all subactions of the `AcceptRequest` choice. Therefore, Canton executes them atomically. In §5, we will show how this ensures liveness and maintains privacy.

4.4 Further operations

`Transfer` may leave the recipient with an endorsement from only $f + 1$ endorsers, as it requires only a quorum of endorsers. As mentioned in §2.3, for liveness the recipient might want $2f + 1$ endorsements. The choice `ExtendEndorsement` enables the recipient to add another validator to any endorsement with at least $f + 1$ validators on it (Listing 3). For authorization reasons, this choice lives in a `MasterAgreement` contract between the validator (signatory) and the user (observer). The qualifier **`nonconsuming`** prevents exercises of the `ExtendEndorsement` choice from deleting the `MasterAgreement` contract, allowing a single `MasterAgreement` contract to be used to extend arbitrarily many `Endorsement` contracts.

```
template MasterAgreement with
    validator: Party
    user: Party
    marketSetup: MarketSetup
  where
    signatory validator
    observer user

    nonconsuming choice ExtendEndorsement: ContractId Endorsement
      with endorsementId: ContractId Endorsement
      controller user
      do
        endorsement <- fetch endorsementId
        assert $ marketSetup == endorsement.marketSetup
        assert $ isQuorum marketSetup endorsement.validators
        exercise endorsementId AddValidators with
          other = endorsement with validators = singleton validator
```

Listing 3: Extending endorsements

Additional choices on `Endorsements` enable merging and splitting as discussed in §2.2. The underlying ideas are similar to `Transfer`, so we omit the details here. In §5.3.1, after having analysed the described implementation, we will present a variation on `Transfer` that hides transfer counterparties even from the validators.

To issue new *CantonCoin*s, a group of validators creates new `Endorsement` contracts for a `Coin` contract after each having checked all of the following:

1. The validator is among the `possibleEndorsers`.
2. The `Coin` contract is active.

Other than creating `MasterAgreement` contracts and issuing new coins, honest validators initiate no other transactions themselves.

5 Analysis

We now show that the implementation of *CantonCoin* from §4 has indeed the properties outlined in §2: safety (§5.1), liveness (§5.2), and privacy (§5.3).

5.1 Safety

As discussed in 2.2, safety for *CantonCoin* has two aspects:

1. Money cannot be created out of thin air.
2. The owner controls when their coin is spent.

Together, they imply that a coin cannot be spent twice; for if it could, then either the total amount of coins would increase or someone else would lose a coin. Conversely, users *can* destroy a *CantonCoin*, e.g., by deleting the `Coin` contract or all of its endorsements. This is the digital analogue of burning bank notes or melting real-world coins.

To ensure that money cannot be created out of thin air, recall the definition of valid coins and the argument in §2.2. We now present the argument more formally, by considering the appropriate *CantonCoin* smart contracts. W.l.o.g., we can fix f to `maxByzantine` in `MarketData`, because the check on line 13 in Listing 1 ensures that all relevant endorsements use the same f.

The money supply for *CantonCoin* is given by the sum of valid coins. In §2.2, we defined valid coins from the users' perspective to be those with $f+1$ valid endorsements. This guarantees that at least one honest validator is endorsing it, even though the user does not know which validators are honest. Our analysis can, however, assume to know who the honest validators are, and we define a valid coin to be represented by an active `Coin` contract endorsed by at least one honest validator in an active `Endorsement` contract. In general, this validator need not be listed in the `possibleEndorsers` of the endorsed `Coin` contract.

We first show that the money supply grows only if honest validators deliberately issue new *CantonCoin*s, independently of what the users or Byzantine validators do. By the definition of valid coins, money supply increases only when new contracts are created: either a `Coin`, or an `Endorsement` with at least one honest validator as a signatory. Assuming that honest validators do not issue new *CantonCoin*s, we show none of these creations increase the money supply, by showing the following two claims:

1. For every created `Endorsement`, either the `Endorsement`'s referenced `Coin` contract was already valid, or a unique coin with the same amount as the `Endorsement` is invalidated.
2. Every created `Endorsement` refers to a `Coin` contract ID that has already been created.

The first claim immediately implies that `Endorsement` creations do not increase the money supply. The second claim ensures the same for `Coin` creations as follows. To increase the supply, a newly created `Coin` requires an `Endorsement` with an honest validator referring to the `Coin`'s contract ID.

Canton ensures that the created contract ID will be fresh (with overwhelming probability), i.e., does not match contract ID of a previously deleted `Coin` contract. So if all `Endorsement` contracts by honest validators refer to contract IDs of already created `Coins`, the new coin cannot increase the money supply.

We prove our claims by examining where `Endorsement` contracts are created in the code.[1] Endorsements are created only as part of coin issuance, `Transfer`, and `AddValidators`.

- For `Transfer`, the two desired claims hold as follows:

    1. Whenever `Transfer` creates an `Endorsement` with at least one honest validator V_h, it always invalidates a unique coin. To see this, note that `Transfer` deletes a `Coin` contract using the `Spend` choice with V_h as one of the choice's controllers. This choice requires V_h to be an observer on the spent `Coin`. The coin then must have been active: V_h checked this as the `Spend` exercise's state predicate during the first phase of the Canton two-phase commit protocol. Thus, `Transfer` indeed invalidates a coin. To see that the coin is unique, note that once the `Coin` is spent, the state predicate check will fail at any other honest validator V'_h who is an observer of the `Coin`.
    2. `Transfer` checks explicitly that the newly endorsed coin is active, and thus created.

- The `AddValidators` choice needs the authority of the `validators` of `other` endorsement. An honest validator V_h does not exercise this choice directly, but only through delegation in `CombineWith` or `ExtendEndorsement`.

    - V_h checks that the `Endorsement` on which `CombineWith` is exercised is still active and that both `Endorsements` reference the same `Coin`. So if the resulting coin is valid, it was already valid before. Moreover, V_h previously endorsed the same `Coin` contract, which therefore must have been created before.
    - The `ExtendEndorsement` adds V_h to the endorsers. It checks that the `Coin` was previously endorsed by at least $f+1$ validators, at least one of which (some V'_h) is honest by assumption. Thus,

[1]We skip the cases for splitting of coins and endorsements and merging of coins, because the analysis is analogous to the cases presented.

the coin's validity does not change. Moreover, V'_h checks that the previous `Endorsement`, on which `AddEndorsement` is called, referred to a previously created contract.

In conclusion, valid coins cannot be created out of thin air, even if users and up to f validators are Byzantine. Finally, the authorization rules also ensure that the owner of a valid coin controls when it becomes invalid. Here, we assume that honest validators delete their `Endorsement` contracts only after the referenced `Coin` contract has been deleted. Then, the argument is the following: The owner is a signatory on their `Coin` contract and the only choice on the contract, `Spend`, has the owner as a controller. The owner's authorization is therefore needed whenever a `Coin` contract is created or deleted. In particular, no group of other users or validators can delete this contract, unless the owner has previously delegated their authorization in a separate contract—i.e., the owner has voluntarily given up control of their coin.

A valid coin need not be usable, though. If there is only one endorser, then the coin cannot be transferred if $f > 0$. In the next section, we show how to ensure liveness for honest owners.

5.2 Liveness

Liveness for *CantonCoin* means that honest owners can always spend their coins, i.e., split them, merge them, or transfer them to other users, independently of Byzantine validators or other third-party users. Here, we assume that honest validators are responsive, i.e., they participate in the Canton commit protocol and send their responses within the given timeouts. We keep the assumption that honest validators delete their `Endorsements` only after the corresponding `Coin` has been deleted. In contrast, a Byzantine validator can delete endorsements and master agreements at any time and they may refuse to respond or send the wrong response. We provide liveness to *honest* owners, i.e., owners who do not themselves delete `Endorsements` of their coins nor deviate from the Canton protocol and the *CantonCoin* DAML code. Note that liveness only makes a guarantee to *honest* users whereas the safety guarantees hold for all users.

First, we establish the following invariant: for each valid coin of an honest user, at the end of each transaction there are at least $f + 1$ active single-validator `Endorsement` contracts by honest validators. Here, a single-validator `Endorsement` is an `Endorsement` contract where the `validators` set is a singleton. Since the user cannot know in advance which validators are Byzantine, this requires $2f + 1$ single-validator endorsements for every coin. We assume that coin issuance is designed to produce $2f + 1$ single-validator endorsements. Coin transfers as described in § 4.3 also produce $2f + 1$ single-validator `Endorsement` contracts for the new coin. As Canton guarantees that the transfer transactions are performed atomically, the invariant holds.

Having single-validator endorsements available enables the owner to (eventually) transfer their coins, as we now argue. To initiate a transfer, the owner must pick $f + 1$ from the $2f + 1$ endorsers. Moreover, the recipient chooses f additional validators that will extend the endorsement. Initially, this choice is random. Then, the owner attempts to transfer the coin as described in § 4.3, namely:

1. The chosen $f + 1$ endorsements are joined,
2. The `Transfer` choice is executed, which deletes the joint endorsement and the old coin and creates a new endorsement with $f + 1$ validators,
3. The `Endorsement` is extended to $2f + 1$ validators and split into $2f + 1$ single-validator endorsements.

If any of these $2f + 1$ validators are Byzantine, then they can cause the transaction to fail. However, Canton will report to the owner the validators that rejected the transaction or did not respond. So, the owner can pick another set of $f + 1$ endorsements without the reported validators, and similarly the recipient for the additional endorsers, and try again. With each failed attempt, they eliminate at least one Byzantine validator. After at most $f + 1$ attempts, the transfer will therefore succeed. This argument hinges on the following properties:

- In Canton, any action on a contract requests responses only from confirmers, who are signatories and authorizers. Other parties' (including observers') responses are not required. In particular, the `possibleEndorsers` of a `Coin` will only be required to an exercise of `Spend` if they are the authorizers, which means that they are among the $f + 1$ chosen endorsers. Similarly, Byzantine third-party users cannot block transactions because Canton does not involve them. Canton sill guarantees that all `possibleEndorsers` will be informed of the spend, though.
- When the transfer transaction fails, by atomicity, the initial endorsements remain active and single-validator; in particular, they are not

merged. Byzantine validators can delete their endorsements at any time, but this is analogous to eliminating them during a failed attempt.

- Byzantine validators cannot merge their endorsements with other validators' without the owner's consent. We ensure this by making the owner a controller on the choices `AddValidators`, `CombineWith`, and `ExtendEndorsement`. This guarantees that the owner can pick any subset of $f + 1$ validators from its $2f + 1$ single-validator endorsements, at least one of which does not contain a Byzantine validator.
- The constraint $3f + 1 \leq N$ ensures that there are always at least $2f + 1$ honest validators. So the recipient can find f honest validators that are distinct from the $f + 1$ endorsers chosen by the owner.

5.3 Privacy

As described in §3.3, a commit in Canton involves only the informees of a transaction. For example, in a transaction consisting solely of the action in Fig. 2, only Alice, V_1, and V_2 receive any data about the transaction. Other users and validators receive nothing. Moreover, Canton provides *sub-transaction privacy.* That is, given a transaction, the users will see only the actions (including their subactions) of which they are informees. For example, only Alice and V_1 can see the outermost (grey) action in Fig. 2. Since every action includes its consequences, Alice and V_1 learn about the sub-actions (yellow and green) even if they were not informees on them.

A transfer requires creating a joint `Endorsement` contract in the state partition of some $f + 1$ validators, say the set $\mathcal{V}_{f+1}$. As a consequence of the `Transfer` choice (Listing 2), this contract and the old `Coin` contract are deleted, and a new endorsement contract is created, with $\mathcal{V}_{f+1}$ as the backers. The deletion of the old coin happens in the partition of its `possibleEndorsers`, but the creations of the new endorsement are visible only to those in $\mathcal{V}_{f+1}$. The $\mathcal{V}_{f+1}$ validators also learn who is the recipient as the new endorsement contains this piece of information.

For liveness, as discussed in §5.2, the new endorsement is then extended to f additional validators. As these validators are contained in the `possibleEndorsers` of the new `Coin` contract, they see the creation action for the new coin, too. However, they do not directly learn the remitter's (sender's) identity. Clearly, if some of the f new endorsers were also part of the `possibleEndorsers` of the old coin, they can, however, implicitly connect the transfer to the remitter, as they have seen the deletion of the old coin in the same transaction, and know the amounts. To prevent

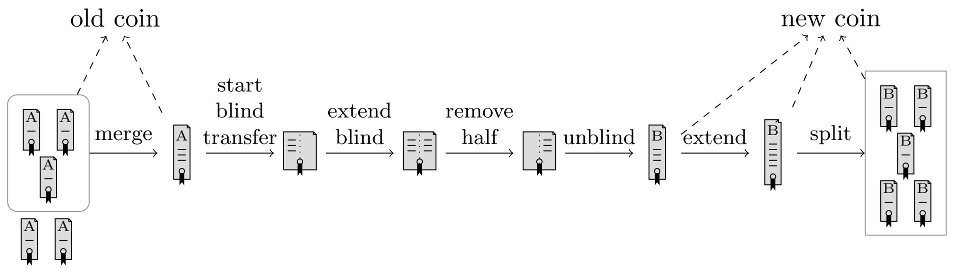

Figure 6: Steps of a blinded transfer for $f = 2$

this, the parties to the transfers may use f fresh validators, which, assuming that the remitter's coin also had $2f+1$ validators, brings the total of involved validators to $3f+1$, up from $2f+1$ if validators are reused. Thus there is a trade-off: spreading the information over more validators, where the validators learn fewer details, or among fewer validators where the validators learn more details. We will see a more drastic example of this trade-off in the next section.

The domain, as discussed in § 3.3, learns the shape of the transaction and the informees. If it has knowledge of *CantonCoin* code, the domain might observe that the shape of the transaction (or its part) fits the shape of a transfer. Assuming that the validators are public, the sequencer can then deduce the counterparties of the transfer from the involved participants. The contents—including the amounts—will, however, remain inaccessible to the domain, as they are encrypted.

5.3.1 Blinded transfers

In §2.4, we describe the idea of how sub-transaction privacy can be used to hide the counterparties of the transaction from any single validator. We now sketch how we implement the idea, by extending the basic model shown in §4. Figure 6 illustrates the steps that the endorsements go through.

The transfer is initiated as described in §4.3, merging $f+1$ single-validator endorsements into one `Endorsement` with $f+1$ signatories. These endorsements reference the old `Coin` contract. Next, however, the remitter now exercises a new choice `StartBlindTransfer` instead of `Transfer`. This creates a new instance of `BlindEndorsement`, a new template that is similar to `Endorsement`, except that it only has an amount parameter, and no owner or coin parameters. Furthermore, its signatories are split into two sets. The first set (shown on the left) is initialized with the signatories of the `Endorsement` on which `StartBlindTransfer` was called, while the second set is initialized to be empty (shown on the right). Then, fresh endorsers are gradually added to the second set, by exercising an `ExtendBlindEndorsement` choice on the `MasterAgreement`, which is analogous to the `ExtendEndorsement` choice. Once $f + 1$ such fresh endorsers have been added, the `RemoveHalf` choice on the `BlindEndorsement` is used to delete that contract and replace it by one on which the first set of signatories is empty. Finally, the `Unblind` choice is executed on the resulting `BlindEndorsement`, where the coin and the beneficiary (as the owner) are specified, resulting in a regular `Endorsement` for that contract. This `Endorsement` can then be extended to f more validators as described in §2.3. These validators must be fresh, i.e., they must not have been endorsers of the remitter's coin. Assuming $2f + 1$ validators on the remitter's coin, executing a blind transfer requires $4f + 2$ live validators.

The `BlindEndorsement` contracts impact the safety and liveness analysis minimally; the total money supply now must also include the amounts in `BlindEndorsement` contracts which have at least one honest validator. The code ensures that honest validators don't issue such endorsements unless f other validators are also signatories on such endorsements, and the original coin is spent.

The confidentiality analysis is straightforward: the only actions on `Endorsement` and `BlindEndorsement` contracts that the $2f+1$ remitter-side validators are informees on are the ones up to the exercise of the `RemoveHalf`. The beneficiary's identity does not appear as part of any contracts or actions in that part of the transaction. Similarly, the beneficiary-side validators are informees only on actions on `BlindEndorsement` and `Endorsement` contracts after the exercises of `ExtendEndorsement` start occurring, and the remitter does not appear as part of any contract or action preceding these exercises.

In Canton, the domain still learns the shape of the transaction and the informees, and might be able to guess the counterparties of the transfer. As before, amounts and other information remain inaccessible to the domain.

6 Related Work

We make a direct comparison between *CantonCoin* and mainstream cryptocurrencies of interest, namely: Bitcoin, Ethereum, ZCash, Monero, Libra, and Ripple. As noted in the Introduction, this paper concerns scalability and privacy. We will focus our comparison on these properties.

6.1 Scalability

Many of the cryptocurrencies relevant to our work rely on proof-of-work consensus, which has inherent scaling limitations [27]. Bitcoin and Ethereum, for example, top out at about 7 to 15 transactions per second. ZCash and Monero are similarly slow, adding zero-knowledge proofs (ZKPs) to the complexity of their proof of work.

Both Libra and Ripple achieve extremely competitive industry speeds. Ripple's consensus algorithm additionally boasts the ability for subnetworks to come to agreement consistently without needing to consult the entire network, thus allowing it to scale horizontally. However, more recent analysis has shown that the algorithm requires these subnetworks to be almost equal to the entire network [7]. Furthermore, consensus on any given subnetwork proceeds in iterative rounds, where the set of possibilities narrows until the subnetwork reaches consensus. Due to the fact that all nodes broadcast messages within the subnetwork, the resulting message complexity makes horizontal scalability with growing subnetworks unlikely without modifications to consensus.

Libra is projected to be quite fast because of its consensus algorithm, *LibraBFT*. LibraBFT has a classical feel, in that it proceeds by leader-election and rounds of voting, which generally leads to prohibitive message complexity as the network scales. However, in LibraBFT the votes and proposals are bundled together in such a way that the communication complexity is linear in the number of nodes on the network (replicas) [26]. This is promising in terms of scalability, as message complexity in such algorithms frequently scales poorly.

However, we note here that by proceeding in rounds and replicating states—as opposed to making distributed commits—both Libra and Ripple lose scalability potential due to the fact that transactions (or blocks) cannot be processed in parallel. More generally, these scaling limits tend to plague proof-of-stake consensus protocols that rule out concurrency by relying on leader elections or proceeding in rounds. Finally, as mentioned in the Introduction, replicating global states creates a total system load that is quadratic in the number of users.

A standard way to deal with this problem is by sharding the state space. For Byzantine fault tolerance, a hybrid with state machine replication can be used, where machines only have to replicate the global state of a shard, as opposed to the entire network. The idea has been applied to permissionless cryptocurrencies in protocols such as ChainSpace[1] and OmniLedger[18] that run shards on top of a BFT consensus algorithm such as PBFT. They use two-step commits—one step to ensure consistency within the shard, the other step to ensure consistency across shards. Furthermore, since shards make commit decisions as a body, each shard has to keep the number of Byzantine nodes under the limit allowed by the consensus algorithm it runs in each shard. As the number of shards increases, usurpation of individual shards by Byzantine nodes becomes more likely, jeopardizing the trustlessness goals. OmniLedger tries to mitigate this problem by rotating the constituents of each shard randomly, recommended on a daily basis. ChainSpace mitigates this issue by killing any transactions in a dispute. Canton also shards the state space, where the shards are denoted by sets of participants. However, unlike ChainSpace or OmniLedger, Canton is permissioned, the shards are not public, and the assignment of data to shards depends on the data, which provides privacy for *CantonCoin*. Canton's domains can also be viewed as another sharding dimension; transactions run over separate domains run in parallel, even if they change the same state partitions (local ledgers). Together, these two forms of sharding allow for true horizontal scalability.

6.2 Privacy

As we have pointed out previously, of the cryptocurrencies present in our analysis, most give few meaningful privacy guarantees. When utilised for a public blockchain, the entire ledger is visible to all participants, something that *CantonCoin* was explicitly designed to avoid. The only mainstream cryptocurrencies with extensive privacy guarantees are ZCash [14] and Monero [20]. However, they are designed largely with anonymity in mind. For companies who may want to transact with the cryptocurrency but want to do so both privately and legally, this may in fact be too strong to be useful, as anonymity may interfere with the standard regulatory checks such as Know Your Customer.

Furthermore, unlike most other blockchains, ZCash and Monero's privacy model prevents them from being auditable. An implementation bug or a cryptographic flaw would be extremely difficult to uncover. This differs from ledgers such as Libra or Ripple, where, barring unfavorable network partitions, users can detect (though not prevent) if more than f validators misbehave to start allowing double spends or creating money. In *CantonCoin*, since the global state can be reconstructed from the individual validators, an audit would require information from each of the validators and undermine users' privacy. We believe that a Libra/Ripple level of auditability can be restored in *CantonCoin* without sacrificing privacy, by having the validators periodically publish structured joint commitments to the total money supply, as in the "synchronization tree-structures" of NOCUST [17].

Another disadvantage of *CantonCoin* is that a user's ability to guard their privacy is inversely proportional to their risk tolerance, as the parameter f factors into the size of the validator set on any given transaction. As mentioned before, blind transaction require the network to have at least $5f + 2$ validators. If a user chooses a more conservative f, such transactions may become impossible. Hence, there is a trade-off between safety and privacy. Finally, users should be aware that domain operators can see message traffic and, if malicious, may be able to perform traffic analysis and deduce information about the users operating on their domain.

7 Conclusion

Today, cryptocurrencies with distributed (but not fully decentralized) trust are built using global ledgers with state machine replication. With *CantonCoin*, we have demonstrated how to build such a cryptocurrency on top of a distributed commit protocol instead. In this way, we were able to get a system with improved privacy guarantees:

1. The ledger is not globally visible to all users;
2. Individual transactions are only visible to subsets of validators (the system's trusted parties); and
3. The counterparties of a transaction are hidden from any individual validator, using Canton's sub-transaction privacy features.

While (1) could be achieved with state machine replication in a closed system of validators, (2) and (3) cannot. Furthermore, while we have not performed any benchmarks due to the alpha state of Canton at the time of writing of this paper, unlike systems with global ledgers, *CantonCoin* can also scale horizontally. In particular, unrelated transactions can be processed using different domains, and thus completely in parallel.

Future Work. We have discussed neither the monetary policies and economic incentives behind *CantonCoin* nor the regulatory compliance aspects in this paper. While the model could work as presented if we assume that the validators do their work pro-bono (as in, e.g., Ripple), we have also experimented with models where the validators impose fees for transactions.

We have also played with changing the money supply, and allowing the users to occasionally mint an interest on the coins they own. Similarly, regulatory rules such as Know Your Customer (KYC) can easily be built as logic in or around `MasterAgreement` contracts in our code base. Our experience shows that Canton is a useful platform for such experiments: the presented model (including blinded transfers) fits in 300 lines of code [6].

We have also not yet considered the case of market setup upgrades, for example adding new validators, or removing validators that decide to leave the network or that misbehave. Technologically, these should be simpler compared to, e.g., Ethereum, as DAML's notion of signatories makes upgrades always possible and makes it clear who must approve an upgrade. On a related note, we believe that the market setup need not be identical among all of the system's users, similar to different UNLs in Ripple's XRP, but we leave this as future work too.

So far, we have created the smart contract code, but not the logic that actually invokes it. Such logic could specify, for example, how the users pick their validators. This would be especially useful if the validators charged processing fees and/or there existed a reputation system for validators (based on their detected Byzantine behavior). In this case, there could be trade-offs between cost and liveness guarantees, or cost and privacy guarantees (e.g., for blind transfers). Also, validators could use external (non-contract) logic to enforce anti-money-laundering rules, for example to monitor the total value of transactions passing through a user's hands. An alternative is to push the system further in the direction of privacy and only show the amounts involved in transfers to an $f + 1$ group of validators, similar to how the described blinded transfers work. The other validators would operate only on cryptographic commitments to amounts.

References

[1] Mustafa Al-Bassam et al. "Chainspace: A Sharded Smart Contracts Platform". In: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* 2018.

[2] Zachary Amsden et al. *The Libra Blockchain.* URL: `https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf`.

[3] Digital Asset. *DAML Programming Language.* URL: `https://daml.com/`.

[4] Sören Bleikertz et al. *A structured semantic domain for smart contracts (extended abstract)*. CSF 2019. 2019. URL: https://www.canton.io/publications/csf2019-abstract.pdf.

[5] Gabriel Bracha and Sam Toueg. "Asynchronous consensus and broadcast protocols". In: *Journal of the ACM (JACM)* 32.4 (1985), pp. 824–840.

[6] *CantonCoin source code*. URL: https://www.dropbox.com/s/6qzijdf1pbqvftl/CantonCoin.daml.

[7] Brad Chase and Ethan MacBrough. "Analysis of the XRP Ledger Consensus Protocol". In: *arXiv:1802.07242 [cs]* (Feb. 20, 2018). arXiv: 1802.07242.

[8] *CoinJoin: Bitcoin privacy for the real world*. URL: https://bitcointalk.org/index.php?topic=279249.0.

[9] Mauro Conti et al. "A Survey on Security and Privacy Issues of Bitcoin". In: *IEEE Communications Surveys & Tutorials* 20.4 (2018), pp. 3416–3452. ISSN: 1553-877X, 2373-745X. arXiv: 1706.00916.

[10] Canton Team Digital Asset. *Canton: A Private, Scalable, and Composable Smart Contract Platform*. URL: https://www.canton.io/publications/canton-whitepaper.pdf.

[11] Arthur Gervais et al. "Is Bitcoin a Decentralized Currency?" In: *IEEE Security & Privacy* 12.3 (May 2014), pp. 54–60. ISSN: 1540-7993. DOI: 10.1109/MSP.2014.49.

[12] Steven Goldfeder et al. "When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies". In: *Proceedings on Privacy Enhancing Technologies* 2018.4 (2018), pp. 179–199.

[13] Bitfury Group and Olaoluwa Osuntokun. *Flare: An Approach to Routing in Lightning Network*. URL: https://bitfury.com/content/downloads/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf.

[14] Daira Hopwood et al. "Zcash protocol specification". In: *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep.* (2016).

[15] Visa Inc. *Visanet – The technology behind VISA*. https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/visa-net-booklet.pdf. 2013.

[16] George Kappos et al. “An empirical analysis of anonymity in zcash”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 463–477.

[17] Rami Khalil and Arthur Gervais. *NOCUST – A Non-Custodial 2nd-Layer Financial Intermediary*. URL: https://eprint.iacr.org/2018/642.

[18] Eleftherios Kokoris-Kogias et al. “Omniledger: A secure, scale-out, decentralized ledger via sharding”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[19] C. Mohan, R. Strong, and Shel Finkelstein. “Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors”. In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983, pp. 89–103.

[20] *Monero*. URL: https://www.getmonero.org.

[21] Pedro Moreno-Sanchez, Muhammad Bilal Zafar, and Aniket Kate. “Listening to Whispers of Ripple: Linking Wallets and Deanonymizing Transactions in the Ripple Network”. In: *Proceedings on Privacy Enhancing Technologies* 2016.4 (Oct. 1, 2016), pp. 436–453. ISSN: 2299-0984. DOI: 10.1515/popets-2016-0049.

[22] Malte Möser et al. “An Empirical Analysis of Traceability in the Monero Blockchain”. In: *Proceedings on Privacy Enhancing Technologies* 2018.3 (June 1, 2018), pp. 143–163. ISSN: 2299-0984. DOI: 10.1515/popets-2018-0025.

[23] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[24] Kari Paul. “US lawmakers hammer Facebook exec over Libra’s threat to privacy”. In: *The Guardian* (July 17, 2019). ISSN: 0261-3077.

[25] *Ripple*. URL: https://www.ripple.com.

[26] *State Machine Replication in the Libra Blockchain*. URL: https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf.

[27] *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*. URL: https://hal.inria.fr/hal-01445797/document.