# Can We Share the Same Perspective? Blockchain Interoperability with Views



SUBMISSION DATE / POSTED DATE

08-06-2022 / 24-10-2022

CITATION

Belchior, Rafael; Torres, Limaris; Pfannschmid, Jonas; Vasconcelos, André; Correia, Miguel (2022): Can We Share the Same Perspective? Blockchain Interoperability with Views. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.20025857.v3

DOI

10.36227/techrxiv.20025857.v3

# Can We Share the Same Perspective? Blockchain Interoperability with Views

Rafael Belchior[a,b,*], Limaris Torres[b], Jonas Pfannschmidt[b], André Vasconcelos[a], Miguel Correia[a]

[a] *INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal*
[b] *Blockdaemon Ltd., Dublin, Ireland*

**Abstract**

With the evolution of distributed ledger technology (DLT), several blockchains have appeared that provide privacy guarantees, including Corda, Hyperledger Fabric, and Canton. These distributed ledgers only provide partial consistency, meaning that participants can observe the same ledger differently, i.e., observe some transactions but not others for privacy reasons. However, this lack of access to all transactions may hamper the development of applications that depend on reasoning about shared state.

We propose to use the concept of *view* – an abstraction of the set of transactions a participant can access at a certain point of time – to systematically reason about the state of blockchains, even if they provide privacy. We introduce BUNGEE (Blockchain UNifier view GEnErator), the first DLT view generator, to allow capturing snapshots, constructing views from these snapshots, and merging views, according to a set of rules specified by the view stakeholders. Creating views and operating over views allows new applications, such as stakeholder-centric snapshots for audits, cross-chain analysis, blockchain migration, and combined on-chain-off-chain analytics. An important subset of these applications that we cover in the paper is related to *blockchain interoperability*.

*Keywords:* Distributed Ledger Technology, Blockchain, Interoperability, Cross-chain, Protocol, Views

## 1. Introduction

Blockchains or DLT (we use these terms interchangeably) provide trustworthy and transparent services, leveraging a network of mutually untrusting participants. A highly desirable property of DLTs is *consistency* [1]: the guarantee that all honest parties share a common prefix of the blockchain, i.e., see the same *transactions* registered in the ledger. Based on this property (which is also an assumption about how blockchains operate), each ledger holds a single source of truth for all its *participants*[1]. Consistency is the foundation of the decentralized trust that DLTs offer.

Despite the importance of consistency, some permissioned DLTs offer only *partial consistency*, providing a trade-off between transparency and privacy. Partial consistency is a weaker notion of consistency that implies that honest parties are able to read only subsets of the same global transaction graph, i.e., of the ledger. For every transaction ID a set of parties share, they also agree on the contents and dependencies of such transaction [4]. Partial consistent blockchains are very useful in the corporate context, where privacy and accountability are easier to enforce [5].

Organizations working on/with Blockchains providing partial consistency have been putting resources to enable *interoperability* with other blockchains, following the growing tendency of the space to accommodate DLTs offering different properties and features [6]. The situation becomes more complicated when bridges connect composable smart contracts [7].

A problem naturally emerges from a multi-chain ecosystem: since participants might have different views of a chain, see different data partitions, how to have a consistent view of it from the perspective of a third-party blockchain that we want to interoperate with? In particular, how do we do so if that blockchain is private or provides only partial consistency?

All these questions require handling cross-chain state. Our work bridges the existing gap in making blockchains provide partial consistency and interoperate with other blockchains. We believe that the concept of blockchain *view* is the answer to this problem. A view offers a stakeholder-centric, generalizable, self-describing commitment to the state of a blockchain, allowing for representing states from different blockchains in a standardized way.

In this way, views help an external observer to reason about partial consistency in DLTs. On the other hand, in public DLTs with probabilistic consensus, temporary conflicting views on the same ledger exist (forks), but are resolved by some mechanism (e.g., the longest chain rule), and thus consistency holds. The notion of view is also beneficial for such blockchains (e.g., Bitcoin, Ethereum, NEAR, Polkadot, Cosmos) because it provides a standardized data format (the view) that can be used for interoperability purposes.

*Corresponding Author:
*Email address:* rafael.belchior@tecnico.ulisboa.pt (Rafael Belchior)

[1] A similar assumption could be extended to business processes, where a single model can capture simple processes. However, different representations of the same process are possible as soon as its complexity increases. The concept of view has its roots in database schema integration and, more recently, business process view integration [2]. To account for the multitude of business process views, *business view process integration* (BVPI) studies the consolidation of different views regarding a business process [3, 2]. Business view process integration (BPVI) addresses the challenges of processes that involve several participants with different incentives, alleviating them by merging models that represent a different view of the same model.

In this context, building and analyzing views is important to accurately understand each stakeholder's view of each DLT as a tool for *interoperability*, but also for business intelligence (e.g., for better understanding a protocol) or auditing (e.g., monitoring a protocol) [8]. Views directly support blockchain interoperability since it is easier to share the perspectives of all participants across heterogeneous DLTs [6, 9], allowing a better representation of the business ecosystem. This could enable complex orchestration of cross-blockchain services and support the new research areas of DLT interoperability, including blockchain gateway-based interoperability [10, 11, 12] and general-purpose interoperability [13, 14].

This paper proposes the *Blockchain UNifier view GEnErator* (BUNGEE), the first system that creates, merges, and processes DLT views. The views are generated in two major steps: 1) taking a snapshot of the blockchain states according to a specific participant, and 2) constructing the view considering the desired time interval. After different view generators create partial views, a merge operation may be applied to the views to produce an integrated view, according to a *merging algorithm*. Since BUNGEE can integrate views, it may be considered a view integration system [15, 2, 16]. We focus our contributions on two research questions (RQ):

*RQ 1. How to generate blockchain views?* Multiple DLT data formats result from their architecture, consensus, and identity models. Formalizing the blockchain view and related concepts is necessary to reason about data representation across chains. We first present a formalization of concepts surrounding the view, rooted in the state abstraction and causality relationships between transactions, states, and views. This formalization is the foundation to describe BUNGEE's algorithms to generate a view. BUNGEE is a flexible, modular middleware that sits between the data and the semantic layers of a blockchain, allowing data to be abstracted into different data models and formats. To the best of our knowledge, this is the first time views are used as a mechanism to take stakeholder-specific snapshots of the ledger, allowing several applications.

*RQ 2. How can one merge views and create an integrated view?* Creating views allows one to see the perspective of a stakeholder over the entire ledger. However, how do we obtain a holistic view of a DLT providing partial consistency, i.e., combined perspectives of all participants? We ensure that the view's creation, merging, and processing come with integrity and accountability guarantees. As a core contribution of this paper, we present the algorithms to merge and process a view, providing a comprehensive discussion of decentralization, efficiency, and privacy trade-offs.

By answering the proposed research questions, we expect to analyze, model, design, and provide implementation guidelines for systems generating views, so that it becomes easier to reason about systems interacting with several blockchains.

*Paper Outline*

This document is organized as follows: In Section 2 we introduce a collection of concepts around BUNGEE. Next, Section 3 presents BUNGEE, including the system model (Section 3.1), the snapshotting phase (Section 3.2), the view building phase (Section 3.3), the view processing phase (Section 3.4). After that, we present the discussion (Section 4). Next, we present the related work, in Section 5. Finally, we conclude the paper (Section 6).

## 2. Blockchain Views

This section introduces a running example that applies view integration to the supply chain industry. After that, we present a formalization of concepts related to the view, such as the access point, blockchain view, and view generator.

### 2.1. Motivation Case Study

We present a typical use case on private blockchains, supply chain [17], that benefits from representing the various internal views to an external observer.


Figure 1: Supply chain scenario with five participants engaging in asset trading.

A supply chain transfers value between parties, from the raw product (physical or intellectual) to its finalized version. Managing a supply chain is complex because it includes many non-trusting participants (e.g., enterprises and regulators). As many markets are open and fluid, companies do not take the time to build trust and, instead, rely on a paper trail that logs the state of an object in the supply chain. This paper trail is necessary for auditability and can typically be tampered with, leading to the suitability of blockchain to address these problems by monitoring the execution of the collaborative process, ensuring that the execution of the process complies with business rules [18, 19]. Audits inspect the trail of transactions referring to a product's lifecycle. Therefore, different perspectives might need to be analyzed. A challenge naturally emerges: balancing the necessary transparency for audits while maintaining privacy about the transactions across other business partner groups is not trivial. By selectively sharing a common domain, parties can have more efficient processes while performing data-sensitive operations within the same supply chain. A domain is the state shared by parties enrolled in a private relationship. Views can then be merged according to custom rules (respecting privacy needs) for auditing purposes.

Let us consider a group of five organizations on a Hyperledger Fabric blockchain that produce, transport, and trade, as illustrated by Figure 1:

- A Supplier, producing goods.
- A Shipper, moving goods between parties.
- A Distributor, moving goods abroad. Buys goods from Suppliers and sells them to Wholesalers.
- A Wholesaler, acquiring goods from the Distributor.
- A Retailer, acquiring goods from shippers and wholesalers.

The Distributor may prefer to make private transactions with the Supplier and the Shipper to keep confidentiality towards the Wholesaler and Retailer (hiding their profit margins). Conversely, the Distributor may want a different relationship with the Wholesaler. It charges them a lower price than it does with the Retailer (as it sells assets in bulk). The Wholesaler may want to share the same data with the Retailer and the Shipper (because the Wholesaler may charge Retailers a higher price than the Shipper). These private relationships are our use case's *domains*.

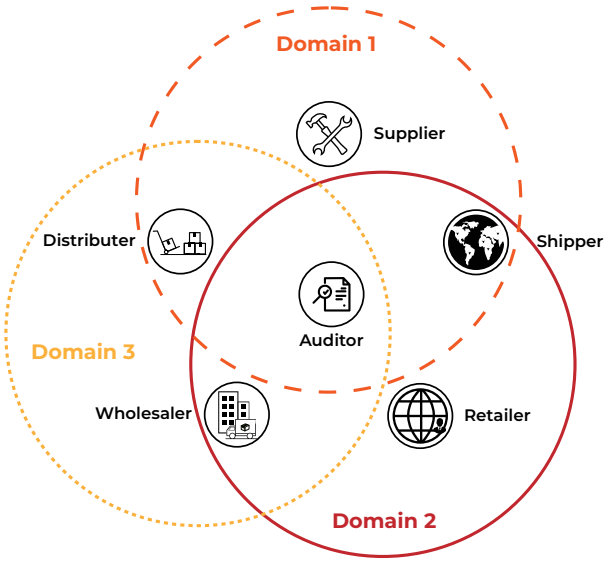


Figure 2: Different domains on the blockchain supporting the supply chain scenario. For instance, the Supplier has access to domain 1, while the Shipper acesses domains 1 and 2. Access to different domains leads to the creation of different views.

Domains hold a subset of the ledger that is only accessible by authorized parties. By sectioning the shared ledger, different views on the same blockchain are possible, depending on a stakeholder's participation in a given domain, as shown in Table 1. In this table, the asset ID is one across all domains. However, its price differs across domains, translating into different views. For instance, the Supplier has access to the asset's price on $d_1$, but only access to its price's hash on $d_2$ and $d_3$ (i.e., does not have access to the price on $d_2$ and $d_3$. This three-dimensional tuple access-deny-deny corresponds to $v_1$. Retailers' view, $v_5$ can see the price of the asset only in $d_2$ . The three existing domains (see Figure 2, translate into three different price values for the same item – five participants originate five different views.

Let us now assume that we identify each asset tracked in a supply chain by an ID and a price. For the same asset (and thus the same state on the blockchain, as it is uniquely identifiable by its ID), the Distributor-Supplier-Shipper (Domain 1, $d_1$) has the same view of the price, but the Wholesaler-Retailer-Shipper (Domain 2, $d_2$) and Distributor-Wholesaler (Domain 3, $d_3$) and have different views. As every stakeholder has a different combination of the domains that are accessible, the DLT infrastructure yields five different views.

| Participant\Domain | $d_1$ | $d_2$ | $d_3$ | |
|---|---|---|---|---|
| Supplier | ID: 1<br>Price: 1 | ID: 1<br>Price: hidden | ID: 1<br>Price: hidden | $v_1$ |
| Shipper | ID: 1<br>Price: 1 | ID: 1<br>Price: 2 | ID: 1<br>Price: hidden | $v_2$ |
| Distributor | ID: 1<br>Price: 1 | ID: 1<br>Price: hidden | ID: 1<br>Price: 3 | $v_3$ |
| Wholesaler | ID: 1<br>Price: hidden | ID: 1<br>Price: 2 | ID: 1<br>Price: 3 | $v_4$ |
| Retailer | ID: 1<br>Price: hidden | ID: 1<br>Price: 2 | ID: 1<br>Price: hidden | $v_5$ |

Table 1: Participant views on the supply-chain blockchain regarding an asset with ID = 1.

Let us now imagine that an auditor wants to inspect the Distributor's operations regarding a good. The auditor would retrieve snapshots of the blockchain in light of each participant's view. After that, the auditor can analyze each view from the perspective of each participant. If a general picture is needed, all views can be merged into an integrated one and jointly analyzed. Since there are different viewpoints, there are different prices for the same object and different merge procedures are possible. The first would be to reveal only one price corresponding to one of the views; a second option would be to show all prices corresponding to all views (better suited for an audit); a third option would be to hide all prices (so the only information that the auditor sees is that prices are different). In particular, the different views are translated into an integrated view that only refers to a consolidated price as a summary of the prices of the different views. The processing and merging of the views are the consortium's responsibility for managing the blockchain, and thus several options are possible. We will address this point later in this paper.

### 2.2. Formalizing Views

This section formally defines the necessary terms for blockchain view integration. We provide the conceptual framework to build programs that can merge blockchain views. The first concept of our framework is the ledger. A ledger is a simple key-value database with two functionalities: `read` and `store`. It supports a state machine that implements a DLT. We define ledger as follows:

**Definition 1.** *Ledger.* A ledger $\mathcal{L}$ is a tuple $(\mathcal{D}, \mathcal{A})$ such that:

- $\mathcal{D}$ is a database, specifically a key-value store. Each entry in the database is a key-value tuple, i.e., $d \in \mathcal{D} : (k, v)$, where $k$ stands for key and $v$ for value.

  $\mathcal{D}$ has two functions: `read` and writes (storing). `read` returns the value associated with a key, the empty set $\emptyset$ (if there is no value for that key) or an error $\perp$ (if the user does not have access permissions), i.e., `read` $\rightarrow \{v, \emptyset, \perp\}$. The `store` primitive saves the $(k, v)$ pair in the database, indexed by $k$, returning 1 if the operation was successful and 0 otherwise, i.e., `store` : $k \times v \rightarrow \{0, 1\}$.

  The `read` and `store` primitives support the representation of simple UTXO blockchains (e.g., Bitcoin) [20] or more complex ones, an account model (e.g., Ethereum) [21], or others by combining the operations mentioned above (e.g., Hyperledger Fabric [22]).

- $\mathcal{A}$ is an access control list that specifies access rights to read entries from the database. Each entry of the list has the form $(p, k)$. Each entry indicates that participant $p$ can read the state with key $k$. A participant $p$ can access a key $k$ when the primitive `access`$(\mathcal{A}, p, k)$ returns 1, or 0 otherwise.

The simple functionality of the notion of ledger given in Definition 1 allows us to represent Bitcoin [23] as follows: the database (collection of all states) is a list of UTXO entries (states). A UTXO has a unique identifier, the transaction hash, and the state key (we present a simplified version of UTXO). Its value is in the form (input, output, metadata). The input corresponds to a reference to the previous transaction and a key to unlock the previous output to the current input. The output consists of a cryptographic lock and time. Metadata is any other relevant information for a transaction using that UTXO (for example, the timestamp and the fees). Hyperledger Fabric's state is more straightforward to map since it is a key-value store.

We define the entities that can read or write on the ledger by *participants*:

**Definition 2.** *Participant.* A participant $p \in \Upsilon$ is an entity ($K_k^{id}$, $K_P^{id}$, id), capable of reading and writing to a ledger L, where:

- $K_k^{id}$ is a private key. The private key is used as the signing key.
- $K_P^{id}$ is a public key. The public key is used as the verifying key.
- id is the unique identifier of the participant. It is the output of a function over the participant's public key[2].

Participants interact with the ledger via nodes. Nodes are software systems that participate in the ledger consensus by aggregating and executing transactions and sending them to other peers (for example, miners in proof-of-work blockchains or peer nodes in Hyperledger Fabric). Participants need a node client (or simply a node) to read data or transact on DLTs (by redirecting signed transactions to them). We introduce the concept of *Access Point* to formalize the relationship between participants and nodes as follows:

**Definition 3.** *Access Point (AP).* An AP $\omega$ maps a node $n$ connected to ledger $\mathcal{L}$ to a set of participants $p_n$, i.e., $\omega(n) \longrightarrow p_n \subseteq \Upsilon_{\mathcal{L}}$. Conversely, $\omega^{-1}$ returns the node set $n_p$ that a participant can access, i.e., $\omega^{-1}(\upsilon) \rightarrow n_p$.

An access point tells us which participants can access the ledger via a specific node. Nodes can access a DLT via a primitive `obtainDLT`. The result of `obtainDLT`$(n_p) = \mathcal{L}_v$, where $\mathcal{L}_v$ is a *virtual ledger*. A virtual ledger only allows the participants to read and write in the ledger according to their permissions (namely, the defined access control list). In more detail, a virtual ledger:

**Definition 4.** *Virtual ledgers.* A virtual ledger $\mathcal{L}_v$ is a projection of a ledger $\mathcal{L}(\mathcal{D}, \mathcal{A})$ in the form $(\mathcal{L}, \mathcal{F}_\pi)$ such that:

- $\mathcal{L}$ is the ledger that provides the database where projections are made.
- $\mathcal{F}_\Pi$, a set of projection functions $\{\mathcal{F}_{\pi_1}, \mathcal{F}_{\pi_2}, ..., \mathcal{F}_{\pi_n}\}$ that returns a subset $d_\pi$ of the database $\mathcal{D}$ from $\mathcal{L}$, i.e., $\mathcal{F}_\pi \in \mathcal{F}_\Pi$: $\mathcal{L}_\mathcal{D} \times \mathcal{L}_\mathcal{A} \times p \rightarrow \{\emptyset, d_\pi\}$, according to the participant's access control list entries defined by $\mathcal{A}$ (or $\emptyset$, if the participant is not authorized to access the ledger). This corresponds to "what the participant can see".

Let us recall that the database or a subset is a collection of keys and their values. We can simplify its representation by referring to the projection of ledger $l$ against participant $p$ (this is, the projection function $\mathcal{F}_\pi$ that is chosen projects the states of the virtual ledger that are accessible by $p$). The projection on ledger $\mathcal{L}$ using the projection function $\mathcal{F}_p$ outputs a set of states $\{s_1, ..., s_n\}$ that participant $p$ can access:

$$d_{\mathcal{L},\mathcal{F}_p} = \{s_1, ..., s_n\}$$

We use the notation $\overline{s}$ to represent the absence of a state in a projection. Consider the following projections, illustrated by Table 2:

- $d_{\mathcal{L}\mathcal{F}_{p_1}} = \{\ s_1,\ \overline{s_2},\ \overline{s_3}\ \} = s_1$
- $d_{\mathcal{L}\mathcal{F}_{p_2}} = \{\ s_1,\ s_2,\ \overline{s_3}\ \} = s_1, s_2$
- $d_{\mathcal{L}\mathcal{F}_{p_3}} = \{\ s_1,\ \overline{s_2},\ s_3\ \} = s_1, s_3$
- $d_{\mathcal{L}\mathcal{F}_{p_4}} = \{\ \overline{s_1},\ s_2,\ s_3\ \} = s_2, s_3$
- $d_{\mathcal{L}\mathcal{F}_{p_5}} = \{\ \overline{s_1},\ s_2,\ \overline{s_3}\ \} = s_2$
- $d_{\mathcal{L}\mathcal{F}_{p_6}} = \{\ \overline{s_1},\ s_2,\ \overline{s_3}\ \} = s_2$

[2]For instance, Bitcoin addresses are used to uniquely identify Bitcoin accounts and are formed by double-hashing the public key associated with that account.

| $d_{\pi_{l,pn}}$ | $d_1$ | $d_2$ | $d_3$ |
|---|---|---|---|
| $p_1$ = Supplier | $s_1$ | $\overline{s_2}$ | $\overline{s_3}$ |
| $p_2$ = Shipper | $s_1$ | $s_2$ | $\overline{s_3}$ |
| $p_3$ = Distributor | $s_1$ | $\overline{s_2}$ | $s_3$ |
| $p_4$ = Wholesaler | $\overline{s_1}$ | $s_2$ | $s_3$ |
| $p_5$ = Retailer | $\overline{s_1}$ | $s_2$ | $\overline{s_3}$ |
| $p_6$ = Retailer | $\overline{s_1}$ | $s_2$ | $\overline{s_3}$ |

Table 2: Ledger $l$ projections onto all the participants from the use case depicted in Section 2.1. The projection function is a simple read of the database. A state $s_i$ in the green background is a state that can be accessed by a participant, whereas a state $\overline{s_i}$ is a state that is not accessible for a given participant.

We can retrieve a projection $d_{\mathcal{L},\mathcal{F}_p}$ (or the empty set) with the primitive `obtainVirtualLedger`. This primitive receives as input a ledger $\mathcal{L}$ and a projection function $\mathcal{F}_p$. Some projections are not unique (e.g., $d_{\mathcal{L},\mathcal{F}_{p_5}}$ and $d_{\mathcal{L},\mathcal{F}_{p_6}}$). The concept of projection is the basis for the DLT view, which we will define later in this section.

Both ledgers and virtual ledgers are abstractions to access a state, represented by a key-value store. Participants issue transactions to change the state. A transaction is defined as follows:

**Definition 5.** *Transaction.* A transaction $t$ is a tuple $(t_{id}, t, payload, \sigma_{K_s^p}(message), S_{tid}, target)$, where:

- $t_{id}$ is a unique increasing sequence identifier for a transaction. This identifier allows one to construct a transaction ID, a unique identifier for a transaction. Transaction $t_i$ precedes $t_j$ ,i.e., $t_i \preccurlyeq t_j$ if and only if $j > i$.
- $t$ is the transaction timestamp
- *payload* is the transaction payload. The payload can carry arbitrary information (smart contract parameters, UTXO input value).
- a signature on the transaction $\sigma_{K_s^p}(message)$, where $message \doteq (t_{id}, t, target, payload)$.
- $S_{tid}$, a set of input states given as input to the transaction with transaction ID *sid*.
- *target* is the state id to which the transaction refers.

A transaction takes as input a $S_{tid}$ and outputs $S'_{tid}$. We define a primitive `VerifyTx(.)` that takes a set of initial states, a transaction, and a set of output states, and outputs 1 if and only if the state transition is valid according to some algorithm $\rho$, i.e., $\texttt{VerifyTx}(S_{tid}, t_{id}, S'_{tid}, \rho = 1)$. Checking the validity of a transaction w.r.t. the states it changes implies re-running the transaction on its run environment. Transactions produce state changes. We define the state as:

**Definition 6.** *State.* A state $s$ is a tuple $(s_k, s_{k,v}, \mathcal{T}, \pi_k)$, where

- $s_k$ is a unique identifier (the state's key).
- a transaction list $\mathcal{T}$, referring to that state, i.e., $\forall t \in \mathcal{T} : t.target = s_k$
- the value it holds $s_{k,v}$. The value of a state can be calculated using the set of transactions $\forall i, TS = \{t_i \subset \mathcal{T}, s \in \mathcal{S} : t_i.target = s_k\}$, i.e., the transactions referring to that state, in the following manner:

$$s_{k,v} = \begin{cases} \emptyset & i = 0 \\ apply(t_i, s_{k,i-1}) & i \leq |TS| \end{cases}$$

  where *apply* is a function that executes the payload of the transaction $t_i$ over the state $s_k$. The data is the most recent state value, the result of the successive transformations (over the previous versions of the same state). Function *apply* is blockchain-dependent.

- a proof of state validity $\pi_k = \sigma_{K_s^p}(s_k, s_{k,v}, v)$, where $s_{k,v}$ is the value of state $s_k$ at version $v$, and $\sigma_m$ is the set of signatures of the participants $P \subset \Upsilon$ that creates the proof, over a payload $m$.

A state has a unique reference (or key) $s_k$, and a version $v$ such that when $v$ is updated, it yields $v' > v$. We denote the value pointed by that reference by $s_{k,v}$. If we omit the version, then we refer $s_k$ as the latest value on a certain state. Thus, for all $k \neq k'$, $s_k$ and $s'_k$ represent the latest value of different states. In practice, the value of a state is the result of successively executing transactions over the same object. The value for $s_{k,v}$ or $(s_n)$ can be calculated as follows, where transaction set $\{t_1, ...t_{k-1}\} \in TS$ are the transactions referring specifically to $s_k$:

$$s_{k,0} \xrightarrow{t_1} s_{k,1} \xrightarrow{t_2} \dots \xrightarrow{t_{k-1}} s_{k,v}$$

Each ledger database stores states in its key-value store. The state identifier, $s_k$ is the key, while the tuple $(s_{k,v}, t, \pi_k)$ is the value. Each proof $\pi \in \Pi$ is an object accounting for the validity of the item it describes (transaction, state, view)[3].

Having introduced all the basilar concepts, we can define a DLT view:

**Definition 7.** *View.* A view $\mathcal{V}$ is a projection of a virtual ledger $\mathcal{L}$, in the form of $(v_k, t_i, t_f, p, d_{\pi_{l,p}}, S, \Pi)$, where

- its key $v_k$, is a unique ID
- an initial time $t_i$ and a final time $t_f$ that restrict the states belonging to that view. A view may have no restriction on the temporal interval, i.e., all states that a participant $p$ accesses through $d_{\pi_{l,p}}$ are included in the view.
- $p \subseteq \Upsilon$ is the participant set associated with the view.
- a projection function $d_{\pi_{\mathcal{L},p}}$ used to build the view.
- $S$ corresponds to the set of versioned states that the participant on the view has access to (via the projection function).

[3] In Bitcoin, for instance, the proof of validity for a transaction is the issuer's signature, along with a nonce whereby its hash begins with a certain number of zeroes and is smaller than a certain threshold (valid transaction within a valid block). In Hyperledger Fabric, the proof is a collection of signatures from the endorsing peer nodes that achieved consensus on the transaction's validity.

- $\Pi$ is a set of proofs accounting for the validity of a view (e.g., accumulator value for over states ordered by last update).

The consolidated view, or the global view $\mathcal{V}$, is the set of all participant views, i.e., $\mathcal{V} = \cup_{i=0}^{i} p_i$, that captures the whole ledger $\mathcal{L}$.

## 3. BUNGEE, a Multi-Purpose View Generator

In this section, we present BUNGEE. First, we present the system, key management processes, and adversary models. After that, we present the snapshot process. Next, we present how views are built and then merged. The section finishes by discussing the processes around creating snapshots, views, and merging views.

### 3.1. System Model

We consider an asynchronous distributed system, the DLT, hosting a ledger $\mathcal{L}$. Three types of participants interact with the ledger: i) participants $\Upsilon$: entities that transact on the network (can use `read` and `write` operations) via the nodes that their AP exposes; ii) nodes $\mathcal{N}$, who hold the full state of the DLT, and contribute to the consensus of the later; and iii) view generators $\mathbb{G}$, programs that build views, via node that has access to the target participant of the view. This implies that a view generator trusts the node that is the access point to the DLT.

In public DLTs, participants can generally assume that the information retrieved by nodes is accurate and cannot be tampered with, as it is easy to verify it against other nodes. However, in private blockchains, the verification is not as straightforward, as participants may not have visibility of the internal state of the ledger via other nodes. Each party manages its keys.

*Key management*

Each participant $p \in \Upsilon$, node $n \in \mathcal{N}$, and view generator $\mathbb{G}$ is identified by a pair of keys $(K_p^p, K_k^p)$, $(K_p^n, K_k^n)$, and $(K_p^{\mathbb{G}}, K_k^{\mathbb{G}})$ respectively. The private key is the signing key, while the public key is the verifying key. The generated keys are independent of all other keys, implying that no adversary with limited computational resources can distinguish the key from one selected randomly. We assume keys are generated and distributed in an authenticated channel, preserving integrity; digital signatures cannot be forged. We say an entity $x$ signs a message $m$ with its private key with the following notation: $\texttt{sign}_x(m)$. Verifying a message $m$ with the public key from $x$ can be done with a `verify` primitive, which outputs 1 if the message was correctly signed by $m$ i.e., $\texttt{verify}(m, x, K_x^p) = 1$, and 0 otherwise.

The DLT is assumed to be able to preserve its safety and liveness abilities despite the possible existence of malicious nodes. This implies that building and operating views based on networks that cannot guarantee safety properties (e.g., DLT forks due to attack) are invalid.

*Adversary Model*

The DLT where view generators operate is trusted, meaning that most internal nodes are honest, and thus the network is trusted. Given this assumption, there can be different adversary models for nodes, generators, and view generators. Nodes can be honest by following the DLT protocol, establishing consensus with other honest nodes, and reporting the actual status of the DLT to participants who request it. Nodes can, instead, be malicious, i.e., Byzantine, being able to deviate from the protocol and falsely report the DLT status to participants (endangering the creation of truthful views). Nodes can be malicious but cautious, meaning they are only malicious if there are no accountability checks that can penalize them (i.e., if they know that they cannot get caught).

View generators constitute a trusted group with the participant that is the target of the view because the generator needs the participant's credentials to access a (private) subset of the ledger. We then assume that each participant runs its view generator. View generators can only build views for participants whose keys they do not control if the ledger does not have any partition (i.e., it is public). Since participants might access DLT partitions from different nodes, the trust group (participant, view generator) does not include a node or set of nodes, i.e., view generators and DLT participants are independent of the nodes that sustain the DLT.

*View Generation Process Overview*

BUNGEE constructs views from a set of states from an underlying DLT called a snapshot. This is done by obtaining a virtual ledger, on behalf of a certain participant, with a projection function. Then, each state accessible by the participant is collected in the snapshotting phase. States are processed, and a representation of the ledger that the participant has access to is built. We can think of the snapshot as capturing available transactions from the perspective of participant in a table (*c.f* Figure 3, step (1)), upon having the necessary permissions from the ledger ((2)). Right after that, in the view building phase, a view is built from the virtual ledger that the view generator has access to by temporarily limiting the states one can see ((3)). Views can be stored in a local database, providing relational semantics and rich queries. Views are assured to provide provenance, i.e., BUNGEE can trace each component constituting a view down to the transaction.

After that, the view merging phase (optional) comprises merging views into an integrated one (see Section 3.5). For that, an extended state is created from the states present in each view that share the same key. Following that step, a merging algorithm is applied to the extended state. Finally, each view generator signs the integrated view, which can optionally be published in a public forum. The publication in a public forum can be decided by the participants that generate views (social consensus).

Let us focus on the high-level snapshot generation and view generation processes, as exemplified by Figure 3. In this example, we are building two views $\mathcal{V}_1$ and $\mathcal{V}_2$ (from participant $p_1$ and $p_2$, that capture states whose projection functions are $d_{\pi_{\mathcal{L},2022}}$

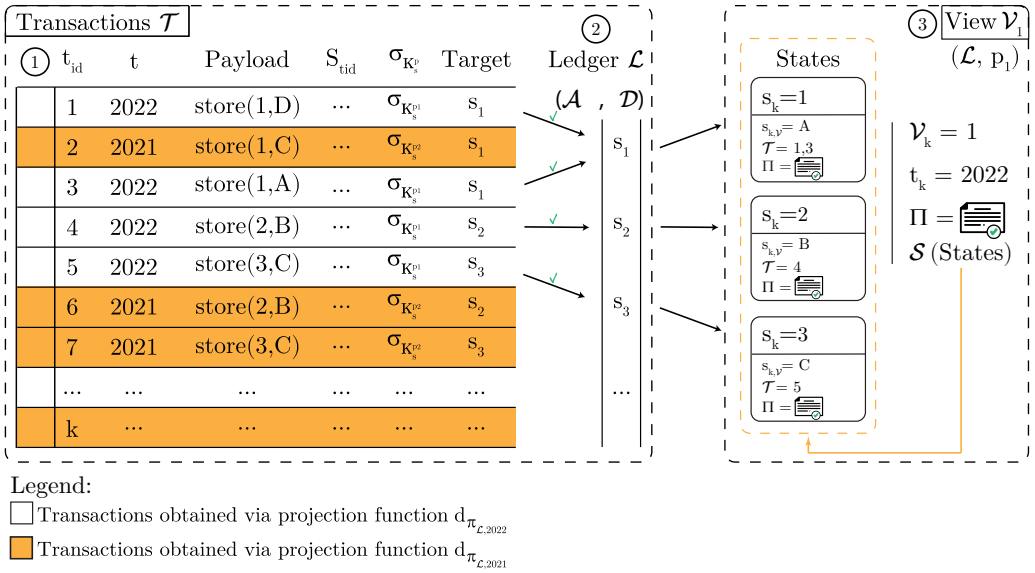


Figure 3: View generation process for participant $p_1$, ledger $\mathcal{L}$, using projection functions $d_{\pi_{\mathcal{L},2021}}$ (yellow rows) and $d_{\pi_{\mathcal{L},2022}}$ (white rows)

and $d_{\pi_{\mathcal{L},2021}}$, respectively. The semantics for the projection functions are simple: $d_{\pi_{\mathcal{L},2022}}$ refers to transactions timestamped as of 2022, and $d_{\pi_{\mathcal{L},2021}}$ refers to transactions timestamped as of 2021. In the Figure, we focus on building $\mathcal{V}_1$ - the white rows. Thus, the transactions timestamped 2021 (in the yellow rows) are not captured on view $\mathcal{V}_1$. Transactions $t_1$, $t_2$ and $t_3$ alter state $s_1$, but $t_2$ belongs to view $\mathcal{V}_2$, so its not included. Transaction $t_4$ changes stores $B$ as the value of $s_3$, while transaction $t_5$ sets $s_3$ as $C$.

Once the three steps are completed, BUNGEE returns the views (the generated and the integrated views) to the client application (for example, a blockchain migration application). For example, the client application might use BUNGEE to retrieve snapshots that refer to a period relevant to an audit. Due to BUNGEE's modularity, adding support for different applications is facilitated. Next, we present each phase depicted in this overview in finer detail.

### 3.2. Snapshot

A snapshot is a set of states that a certain stakeholder can access, plus proof of the validity of that state. We view each state as a versioned (key, value) store. A `snapshot` has a snapshot identifier $id$, a version $v$, a participant $p$, a set of states bins, $sb$, an initial time $t_i$ that refers to the timestamp of the first transaction of any of the states belonging to $sb$, a final time $t_f$ that refers to the timestamp of the last transaction of any of the states belonging to $sb$, i.e., $\texttt{snapshot} \doteq \{id, v, sb, t_i, t_f\}$. Each state bin is indexed by a state id $s_k$, the latest value to that key, $s_{k,\overrightarrow{v}}$, a version $v$ that refers to the number of transactions applied on state key $s_k$ to yield the latest value $s_{k,\overrightarrow{v}}$ and a list of transactions $T$ referring to that state (as in Definition 7). Versioning snapshots allows for efficiently building snapshots from older snapshots (this is, building snapshots from incremental changes from older snapshots).

Algorithm 1 depicts the snapshotting process. The snapshot phase occurs when the BUNGEE client requests the beginning of the view integration process to a node $n$ on behalf of participant $p$ (line 8). After that, the node connects to the DLT. Upon a successful connection, $n$ retrieves the ledger (line 9). Obtaining a list of states from a ledger requires checking all transactions that performed state updates. For each transaction, a BUNGEE has to check its target. BUNGEE creates a new state if there is no state key with a target equal to the current transaction. The version of the new state is one. Then, BUNGEE runs the transaction's payload against the current state value (empty at initialization). Otherwise, if the transaction target refers to an existing state key, run the transaction payload against the state's current value, yielding the new value and incrementing the version by one. This process outputs a list of states. According to the participants' perspective, the process is abstracted by the ledger's projection (according to the participants' perspective) that the algorithm uses (line 11).

The snapshot maps each state to a state bin. For each state, we collect its key (line 15), version (line 16), latest value (line 17), the auxiliary first timestamp (line 18), and auxiliary latest timestamp (line 19). After that, the first and last timestamps are updated (lines 28 and 29), and, at last, the algorithm returns a snapshot.

### 3.3. View Building

This section explains how views are built, answering the research question *How to generate blockchain views?*. A view generator can generate a set of views depending on the input $p$. The following steps occur for each view to be built: first, the view generator generates a snapshot. After that, the snapshot is limited to a time interval and signed by the view generator.

Algorithm 2 shows the process of building a view from a snapshot. First, the view generator temporarily limits each included state, proceeding to abort if no states are within its

**Algorithm 1:** Snapshotting of ledger $\mathcal{L}$ through node $n$ and participant $p$, via projection function $\mathcal{F}_p$

**Input:** Access point $AP$, participant $p$, projection function $\mathcal{F}_p$, snapshot identifier $\texttt{snapshot}_{id}$
**Output:** Snapshot from participant $p$ through node $n$, $\texttt{snapshot}$

1 $\texttt{snapshot.id} \leftarrow \texttt{snapshot}_{id}$
2 $\texttt{snapshot.v} \leftarrow 1$
3 $\texttt{snapshot.sb} \leftarrow \emptyset$
4 $\texttt{snapshot}.t_i \leftarrow \perp$
5 $\texttt{snapshot}.t_f \leftarrow \perp$
6 $t_{it} \leftarrow \infty$ ▷ temporary variable to hold minimum state timestamp to date
7 $t_{ft} \leftarrow 0$ ▷ temporary variable to hold maximum state timestamp to date
8 $n = \omega^{-1}(p)$ ▷ choose any available node
9 $\mathcal{L} = \texttt{obtainDLT}(n)$ ▷ depends on the DLT client implementation
10 $d_{\mathcal{L},\mathcal{F}_p} = \texttt{obtainVirtualLedger}(\mathcal{L}, \mathcal{F}_p)$ ▷ obtain projection of $\mathcal{L}$ according to $p$
11 **foreach** $s_k \in d_{\mathcal{L},\mathcal{F}_p}$ **do**
12 $s_{k,it} \leftarrow \emptyset$ ▷ the timestamp of the first transaction applied to state $s_k$
13 $s_{k,lt} \leftarrow \emptyset$ ▷ the timestamp of the last transaction applied to state $s_k$
14 $\texttt{snapshot}.sb[s_k].s_k = s_k$
15 $\texttt{snapshot}.sb[s_k].version = d_{\mathcal{L},\mathcal{F}_p}[s_k].T.length$
16 $\texttt{snapshot}.sb[s_k].\texttt{latestValue} = d_{\mathcal{L},\mathcal{F}_p}[s_k].s_{k,v}$
17 $\texttt{snapshot}.sb[s_k].T = d_{\mathcal{L},\mathcal{F}_p}[s_k].T$ ▷ save list of transactions referring to each state key
18 $s_{k,it} = d_{\mathcal{L},\mathcal{F}_p}[s_k].T[0]$ ▷ transaction list is ordered chronologically
19 $s_{k,lt} = d_{\mathcal{L},\mathcal{F}_p}[s_k].T.length$
20 **if** $s_{k,it} < t_{it}$ **then**
21 $t_{it} = s_{k,it}$ ▷ update the auxiliary first timestamp
22 **end if**
23 **if** $s_{k,lt} > t_{ft}$ **then**
24 $t_{ft} = s_{k,lt}$ ▷ update the auxiliary last timestamp
25 **end if**
26 **end foreach**
27 $\texttt{snapshot}.t_i = t_{it}$
28 $\texttt{snapshot}.t_f = t_{ft}$
29 **return** *snapshot*

**Algorithm 2:** Constructing a view $\mathcal{V}$ of ledger $\mathcal{L}$ with snapshot $\texttt{snapshot}$, from the perspective of participant $p$.

**Input:** Snapshot $\texttt{snapshot}$, view id $id$, initial time $t_i$, final time $t_f$
**Output:** View $\mathcal{V}$

1 $\mathcal{V}.k \leftarrow id$
2 $\mathcal{V}.t_i \leftarrow t_i$
3 $\mathcal{V}.t_f \leftarrow t_f$
4 $\mathcal{V}.d_{\pi_{l,p}} \leftarrow \texttt{snapshot}.\mathcal{F}_p$
5 $\mathcal{V}.p \leftarrow \texttt{snapshot}.p$
6 $\mathcal{V}.\Pi \leftarrow \perp$
7 $\mathcal{V}.S_{k,v} \leftarrow \perp$
8 **if** $t_i <$ *snapshot*$.t_f$ ***OR*** $t_f >$ *snapshot*$.t_i$ **then**
9 return; ▷ there are no intersecting states that we want to capture, on the snapshot
10 **end if**
11 ▷ each $sb = \{s_k, s_{k,\overrightarrow{v}}, \text{v}\}$
12 **foreach** $s_k \in$ *snapshot*$.sb$ **do**
13 **foreach** $t \in s_k$ **do**
14 **if** $t.timestamp < t_i$ ***OR*** $t.timestamp > t.f$ **then**
15 $\texttt{snapshot}.sb[s_k] \leftarrow \texttt{snapshot}.sb[s_k].\mathcal{T} \setminus t$ ▷ removes transaction that is not within the specified time frame
16 **end if**
17 **end foreach**
18 $\mathcal{V}.S_{k,v} \leftarrow \texttt{snapshot}.sb[s_k]$
19 **end foreach**
20 $\mathcal{V}.\Pi \leftarrow sign_{\mathbb{G}}(\mathcal{V})$
21 **return** $\mathcal{V}$

boundaries (line 8). If there are, each state in the snapshot is included if it belongs to the temporal limit (line 18) and removed otherwise (line 15). Finally, the view generator signs the view (line 20) and returns it to the client application (line 21).

### *3.4. Merging views*

In this section, we describe how to merge views. The merging of views creates an integrated view $\mathbb{I}$ from a set $\mathcal{V}$ of input views. The idea is to compare the state keys indexed by every view and their value according to a merging algorithm $\mathcal{M}$ that is given as input. This merging algorithm controls how the merge is performed.

Algorithm 3 shows the procedure for merging views. The algorithm receives the views to be merged and returns an integrated (or consolidated) view as input. We initialize an auxiliary list $\mathcal{S}_{\mathcal{V}_1,\ldots,\mathcal{V}_n}$ (on line 1) that holds all the values (coming from different views) for each state key. We propose a construct called an extended state. An extended state is a state where each state key maps to a set of values. Additionally, an extended state has a *metadata* field holding a list of operations applied to that extended state.

**Definition 8.** An *Extended State* $\overrightarrow{s}$ is a tuple $\overrightarrow{s_k}, \overrightarrow{s_{k,v}}, t, \pi_k, metadata, version$), where

- $\overrightarrow{s_k}$ is a unique identifier (the state's key);
- $\overrightarrow{s_{k,v}}$ is a list of values;
- a transaction list $\mathcal{T}$;
- a proof of state validity $\pi_k$;
- metadata, which holds a list of operations that have been applied to the extended state;

**Algorithm 3:** Merging a set of views $\mathcal{V} = \mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_n$, where each view was built referring to participant $p_1, p_2, ...p_n$ respectively by a set of view generators $\mathbb{G} = \mathbb{G}_1, \mathbb{G}_2, ..., \mathbb{G}_n$

**Input:** Views to be merged $\mathcal{V} = \mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_n$, merging algorithm $\mathcal{M}$
**Output:** Integrated view $\mathcal{I}$

1 $\mathcal{S} \leftarrow []$ ▷ state list $\mathcal{S}_{\mathcal{V}_1,...,\mathcal{V}_n}$($\mathcal{S}$ for simplicity) where each index (representing a state key) maps to tuple of values from referring to that key, from each view to be merged
2 $\mathcal{I}.t_i \leftarrow \emptyset$
3 $\mathcal{I}.t_f \leftarrow \emptyset$
4 $\mathcal{I}.d_{\pi_{l,p}} \leftarrow \bigcup_{i=0}^{n} \mathcal{V}_n.d_{\pi_{l,p}}$
5 $\mathcal{I}.p \leftarrow \bigcup_{i=0}^{n} \mathcal{V}_n.p$
6 $\mathcal{I}.\Pi \leftarrow \perp$
7 $\mathcal{I}.S_{k,v} \leftarrow \perp$
8 **foreach** $v \in \mathcal{V}$ **do**
9  **foreach** $s \in v.S_{k,v}$ **do**
10  **if** $s \in \mathcal{S}$ **then**
11  $\mathcal{S}[\overrightarrow{s.k}] = \mathcal{S}[\overrightarrow{s.k}] \cup \overrightarrow{s_{k,v}}$ ▷ if state exists, add value referring to that state, from current view
12  $\mathcal{S}[\overrightarrow{s.k}].version \leftarrow \mathcal{S}[\overrightarrow{s.k}].version + 1$
13  **end if**
14  **else**
15  $\mathcal{S}[\overrightarrow{s.k}] = \overrightarrow{s_{k,v}}$ ▷ otherwise, initialize state key list
16  $\mathcal{S}[\overrightarrow{s.k}].version \leftarrow 0$
17  $\mathcal{S}[\overrightarrow{s.k}].metadata \leftarrow \{MERGE - INIT\}$
18  **end if**
19  **end foreach**
20 **end foreach**
21 $\mathcal{I}.S_{k,v} = \texttt{call}_{\texttt{algorithm}\mathcal{M}}(\mathcal{S})$ ▷ OPTIONAL. Computes the state list of the integrated view according to $\mathcal{M}$ (see for example algorithm 5)
22 $\mathcal{I}.d_{\pi_{l,p}} \leftarrow \mathcal{I}.d_{\pi_{l,p}} \cup \{\mathcal{M}\}$ ▷ add reference to the merging algorithm
23 $\mathcal{I}.t_i = \texttt{min}\{\mathcal{I}.S_{k,v}.t_i\}$ ▷ initial timestamp correspond to the initial timestamp of the processed states
24 $\mathcal{I}.t_f = \texttt{min}\{\mathcal{I}.S_{k,v}.t_f\}$
25 $\mathcal{I}.\Pi \leftarrow sign_{\mathbb{G}}(\mathcal{I})$ ▷ signed collectively by $\mathbb{G}$
26 **return** $\mathcal{I}$

- version, a monotonically increasing integer. The counter increases when an update is done to the extended state (the number of elements in the metadata field is the same as the version).

Thus, each index of the set of extended states $\mathcal{S}$ will index all different values for each key for all the views to be merged, i.e.,

$$\mathcal{S}_{\mathcal{V}_1,...,\mathcal{V}_n} = \{\forall s_i \in \mathcal{S} : \exists k_i \in s_i : k_i \implies (s_{\mathcal{V}_1(k_i,v)}, ..., s_{\mathcal{V}_n(k_i,v)})\}$$

After we initialize the list of extended states, in Algorithm 3, we initialize the integrated view properties: its initial timestamp (line 2), final timestamp (line 3), projection functions (taken as the union of the projection functions of all the views, on line 4), participants (the participants from each view, on line 5), a set of proofs (line 6) and a set of states (line 7). The set of states to be assigned as the set of states of the integrated view is a function of the processed auxiliary set of states $\mathcal{S}$. After all, we check each state key to merge each view. If the tested state is already on the auxiliary state set (line 10), then we add its value $\overrightarrow{s_{k,v}}$ as a value for the current extended state key (line 11). This outputs a list of values (between one and the number of views to be merged) for each extended state key. Otherwise, we set a new extended state, adding the current state value (as the first value for that key, on line 15).

On line 21, we apply an optional view processing phase by giving our list of states $\mathcal{S}$ to an arbitrary algorithm that needs to respect a simple interface and functionality (later defined). After that, we add algorithm $\mathcal{M}$ as a projection function for $\mathcal{I}$ for future traceability and auditing. Next, we adjust the initial and final timestamps (lines 23 and 24) because the merging algorithm might have changed the time boundaries of the included states (for example, the state corresponding to the lowest timestamp might have been removed). All the view generators must sign $\mathcal{I}$ (line 25) to promote accountability. Signing the integrated view can be distributed using a multi-signature algorithm (for example, BLS Multi-Signatures [24]).

Each merging phase has an optional application of a merging algorithm $\mathcal{M}$, which dictates how the merge is conducted (otherwise, all states are included without any further processing). We define a simple interface for merging algorithms: a merging algorithm receives a set of extended states as input and outputs a set of extended states. The functionality of the merging functions should be: 1) apply arbitrary operations on the set of extended states, 2) add a reference to the current merging algorithm to the *metadata* field of each extended state key that is altered, 3) increase the version of each extended state key that is altered. Each merging algorithm should be public and well-known to the parties involved.

Examples of merging algorithms are:

- *Selective Join:* keeps certain values from an extended view.

  Algorithm 4 presents the selective join algorithm. This algorithm selects the value by the first view of the view list that is being integrated. In practice, the value for each key that view 1 holds overrides the other values. That state is removed if there is no value for the first view. Applications are similar to join operations in relational databases; selective join allows the view to focus primarily on the ledger state from a perspective of a particular participant while considering others.

- *Pruning:* removes the values coming from a particular view.

  Algorithm 5 prunes the values belonging to a particular view from a set of extended states. Note that times do

NOT need to be updated because those are re-calculated in steps 23 and 24 of Algorithm 3. Applications include removing sensitive information in the context of existing regulations and laws.

**Algorithm 4:** Merging algorithm example – SELECTIVE JOIN (by view $\mathcal{V}_1$)

**Input:** The set of states to be processed $\mathcal{S}$
**Output:** A processed set of states $\mathcal{S}'$

$\mathcal{S}' \leftarrow \emptyset$ ▷
**foreach** $s \in \mathcal{S}$ **do**
  **if** $|s_k| = 1$ **then**
    ▷ if the state key for every view only points to one value, then it means that state is the same for each view
    **continue**
  **end if**
  **if** $\nexists s[0]$ **then**
    **continue** ▷ if there is no value for the first view, do not capture this state
  **end if**
  **else**
    $\mathcal{S}'[s_k] \leftarrow s[0]$ ▷ otherwise, the value for $s_k$ is the first value indexed (belonging to $\mathcal{V}_1$)
    $\mathcal{S}'[s_k].metadata \leftarrow$ `JOIN-VIEW-1`
    $\mathcal{S}'[s_k].version \leftarrow \mathcal{S}'[s_k].version + 1$
  **end if**
**end foreach**
**return** $\mathcal{S}'$

**Algorithm 5:** Merging algorithm example – PRUNE (by $\mathcal{V}_1$)

**Input:** The set of states to be processed $\mathcal{S}$
**Output:** A processed set of states $\mathcal{S}'$

$\mathcal{S}' \leftarrow \emptyset$ ▷
**foreach** $s \in \mathcal{S}$ **do**
  **if** $s_k[0]$ **then**
    $\mathcal{S}'[s_k] = \mathcal{S}'[s_k] \setminus s[0]$ ▷ if there exists a value for view $\mathcal{V}_1$, then remove that value from the state list
    $\mathcal{S}'[s_k].metadata \leftarrow$ `PRUNE-VIEW-1`
    $\mathcal{S}'[s_k].version \leftarrow \mathcal{S}'[s_k].version + 1$
  **end if**
**end foreach**
**return** $\mathcal{S}'$

### 3.5. Example: merging two views

In this section, we graphically show an example of a merge view, by applying algorithms 3 (merge view) and `MERGE-ALL`. Informally speaking, `MERGE-ALL` works by keeping the values from both views included in the final view (a rather simple merge algorithm). Let us consider two views $\mathcal{V}_1$ and $\mathcal{V}_2$ (create from the table of Figure 3), and its merging into a consolidated view $\mathcal{V}_I$, c.f. Figure 4. View $\mathcal{V}_1$ and $\mathcal{V}_2$ differ on the value for $s_1$, A and C, respectively. The integrated view will hold an extended state with 1) a timestamp including both views, 2) references to the participants generating each view, 3) the joint projection function, 4) a set of proofs, and 5) a set of extended states. For $s_1$, we have included the different values from the different views.

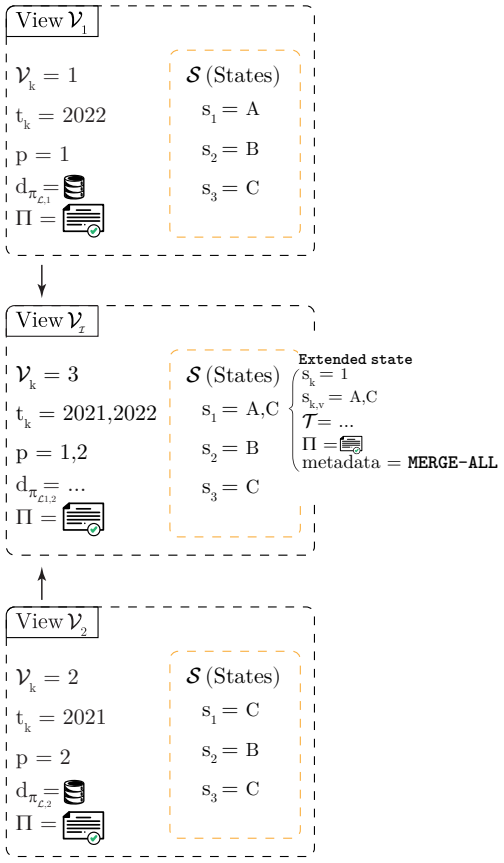


Figure 4: Merging of views $\mathcal{V}_1$ and $\mathcal{V}_2$ into a consolidated view $\mathcal{V}_I$ according to merging algorithm `MERGE-ALL`.

## 4. Discussion

In this section, we discuss BUNGEE. The proliferation of blockchain interoperability solutions is increasing interest in exploring cross-chain logic and the need to model and analyze it [8]. Our proposal constitutes the foundation to make sense of that diversity by allowing to systematically create and integrate views from different blockchains. In this section, we discuss the studied research questions, with considerations on the integrity, accountability, and privacy of views.

### 4.1. RQ 1. How to generate blockchain views?

Views can be generated from different sources as long as they are accompanied by a valid proof. The existence of proofs on states is proof of creation by the set of entities that created or

executed the transactions referring to that state. For example, a signed transaction hash qualifies as proof of a transaction that makes part of the state proof (as many proofs as signed transactions referring to a certain state). On the other hand, views are also signed by the view generator that either generates, merges views, applies a merging algorithm, or notarizes the view as true. This set of proofs allows independent parties to validate the truthfulness of the view (by verifying each state) and hold view generators accountable. In a permissioned environment, an auditor can confirm that the views are valid and complete. The metadata fields on each extended state and the views allow one to understand who, when, and how a view generator changes a certain view. However, more accountability measures can be implemented. In particular, if a view is only shared across the view generators that endorsed it, there might be limited exposure and, therefore, limited transparency. To enhance transparency, our key insight is to store a view in a public forum such as the InterPlanetary File System [25] (a distributed peer-to-peer file system maintained by a network of public nodes) or a public blockchain, similarly to some related work [11, 26]. If a view is deemed false, automatic view conflict detection and resolution can occur.

Besides the integrity availabilities an auditor could impose, the own network could enforce that. For this, we need two conditions to hold. First, each group of nodes that accesses a subset of a ledger (and thus creates a view) must have at least two elements. Second, at least there is one honest element for every group. Thus, an honest view generator connected to an honest node holds knowledge of the view $v$, and publishes it. If a malicious node broadcasts a false view $v'$, an honest node can dispute it. Disputes can be calculated by calculating the difference between views and checking the proofs that constitute each view. In particular, if an instance of BUNGEE, on behalf of participant A, holds the knowledge of a pair of different views $v, v'$ referring to the same participant at the same time frame, then one of the views is false. Thus, the creator of one of the views is malicious. An honest view generator can reconstruct the disputed view and compare it to the view publicized by the malicious participant.

It is unlikely that all participants are colluding to change the perception of the inner state because, in principle, participants have different interests; however, there might be several situations in which the whole network gains if it colludes (i.e., blockchain with financial information). The ledger is unreliable if all internal nodes collude because the safety properties cannot be guaranteed. We hypothesize that using a view similarity metric could be a good tool to assess the quality of the view merging process. In other words, one could systematically compare how the final integrated view is different from each view that composes it.

### *4.2. RQ 2. How can one merge views and create an integrated view?*

In this paper, we have introduced how to create, merge, and process views. However, a challenge remains unsolved: how to share views in a decentralized way? How does one manage the lifecycle of a view, including its creation, endorsement, and dispute? Although the work by Abebe et al. [26] sheds some light on this, how can one verify that a view is false? The solution offered by Abebe et al. includes parties voting on an invalid view, but this does not solve the problem per se because if the source blockchain is private, there is no canonical answer. Suppose that at least one view from the integrated view comes from a private blockchain, the signatures of the view guarantee that a certain participant has voted on the validity of that view. This could introduce problems if all participants collude to show a false view. However, assuming that at least one view generator is honest, the view generator could initiate a dispute with the suspect of a false view.

A view generator could use fraud proofs [27] to create disputes about the validity of views, allowing an efficient and decentralized view management protocol. Application clients can then use the proof field from views, states, and transactions to validate a certain fact on a ledger. However, when BUNGEE merges views, completeness may not be guaranteed because the merged view depends on each input view, and processing might be applied (including pruning), possibly leading to information being excluded. A case to apply pruning might be when sensitive data is recorded on a ledger and later removed from the processing stage or even to remove "obsolete" data from the blockchain and therefore contribute to efficient bootstrapping of light clients [28]. An interesting detail is that each view only includes the state and respective proofs on timeframe $t_k$. However, to ensure that it is possible to validate the view, a pointer to the validity of the latest state before $t_k$ should be available.

Our integration process follows a semantic approach to information based on a conceptual standard data model that we define as a view. Thus, for each practical implementation of BUNGEE, there needs to be a mapping between the data model of the underlying blockchain and the view concept. Being all views uniform, we can not only represent data in all blockchains, but we can merge views belonging to different blockchains. The applicability is building a complete picture of the activity of a participant in each network but can also be used to disclose information according to an access control policy [29]. While selective access control to views has been explored, there is space to explore decentralized identity access control mechanisms to provide fine-grain access over views, leveraging the need to unify the different notions of identity that emerge from different blockchains.

The reader might inquire how BUNGEE would ensure the privacy that partial consistent blockchains attempt to enforce when views are unified and then shared. To address this problem, we envision two solutions: first, merging views requires *tacit* consent from all parties sharing the input views. If there is sensitive data, the data is removed *apriori* or removed in the snapshotting phase. This is essentially encoded by the projection function $\mathcal{F}_p$ used to obtain the virtual ledger. The second solution is to encrypt the data (or hash the data) [29], so the resulting view contains obfuscated information or a notarization proof [30], respectively. However, the scientific community agrees that storing sensitive data on-chain, even if encrypted, is a bad security practice due to the menace of cryptographic algorithms being broken in the future [31, 32, 33, 34]. Zero-

knowledge proofs can also be explored as a vehicle to prove facts on a ledger by disclosing limited information about such facts [35, 36]. We leave those interesting research paths for future work.

*Use cases benefiting from blockchain views*

In this section, we introduce considerations about use cases that can utilize the views we described in our paper. The first use case is cross-chain state creation, management, and visualization. Although some preparatory work has been done [37, 8], it is difficult to visualize and reason about private data partitions (different views), not only in the cross-chain setting but also in a single blockchain setting. Blockchain platforms could leverage views to improve view analysis for auditors, cybersecurity experts, and developers. Auditors and cybersecurity professionals can facilitate audits [38] because different data partitions can be analyzed from a specific perspective. Developers can gain insight into their applications and processes. The representation of on-chain data through a DLT view in multiple chains allows for a visualization of the cross-chain state, making it easier to manage and reason. A specific application could be having one view across multiple Cosmos zones, Polkadot parachains, or Layer 2 solutions (Polygon, Arbitrum, and others, for example) [39].

The second use case is decentralized application migration. Migration of blockchain-based applications is not only necessary, but increasingly common [40, 39, 2]. Migration allows enterprises to experiment with other DLT infrastructures without the risk of vendor lock-in. The key idea behind application migration is to capture the DLT state relevant to that application (data and functionality) and move it to a different DLT infrastructure. We leave the further exploration of this use case for future work.

## 5. Related Work

In this section, we present the related work.

*Partial Consistency*

Blockchain channels limit the information available to each network participant, i.e., channels allow participants to access a subset of the global ledger. Graf et al. propose the concept of partial consistency [4]. While blockchains providing this property have existed for several years, such as Quorum [41], IOTA [42], Corda [43], Hyperledger Fabric, Hyperledger Besu [44], Ripple [45], to the best of our knowledge, this is the first formalization of the concept. Some solutions build partial consistency realizations on top of blockchains [46], such as Digital Asset's Canton [47].

A related concept is *sharding*. Sharding is a technique to improve throughput, typically in public blockchains. A sharding scheme offloads the transaction processing to several groups of nodes called shards [48]. As a result, parallelization is possible, improving throughput and reducing communication overhead between nodes [49, 50]. Therefore, those nodes are only responsible for processing those transactions on their shard. Nodes have different views on the transactions to be processed at the initial stage of the sharding protocol. A shard is thus a logical entity that guarantees the integrity and correctness of states regarding the participants that can access those states. Like a shard, a view is a logical separation of the ledger according to each participant. Our concept of view brings another way to reason about shards, where each validator that is part of a shard runs a view generator and can communicate the shard state to different blockchains.

*Generating views*

Katsis al. [16] has summarized view-based data integration techniques. Our approach follows a Global and Local as View [51] because views are created from a subset of the global state, but then can be merged and processed. We call the reader's attention to the survey on view integration techniques, mostly used in the database and business process management research areas [2]. Abebe et al. [26] have proposed the concept of external view, a construct to prove the internal state of permissioned blockchains. In this paper, the authors show how views can be managed and decentralized while allowing one to prove facts about a private blockchain. We extend and generalize the concept of view so that it can be used for interoperability purposes (by allowing the integration and merging of views). In [29], the authors also use the concept of view as a standardized way to access blockchain data under certain conditions (which we encode in the projection functions), reiterating the need to manage blockchain views in the context of blockchains that provide partial consistency. However, this article does not explain how to use views to share a common perspective across different chains. In [20], the authors propose a general framework for blockchain analytics on the Bitcoin and Ethereum blockchains. However, this work focuses on public blockchains, so the need to merge views is not taken into account.

There are some proposals in industry and academia that propose general data models for cross-chain interaction, namely the Rosetta API, Quant Overledger's gateways [39, 52], Blockdaemon's Ubiquity API [53], Polkadot's XCMP [54, 55], Cosmos's IBC [56]. The Rosetta API and Blockdaemon's Ubiquity API only support public blockchains. Quant Overledger supports public and private blockchains but does not allow them to realize complex operations such as merging views. Polkadot and Cosmos have the previous limitation and can only support blockchains created with Substrate and Tendermint, respectively. On the other hand, BUNGEE aims to create views independent of the underlying blockchains.

*View applications*

In [29], views were used to provide fine-grain dynamic access control over private data in Hyperledger Fabric. In addition to the applications referred to in Section 1, we identify some studies using the concept of view for different purposes. Some authors use views to perform audits of participants on different blockchains [57, 58]. In particular, a view is created and then merged with other views from the same participant on different blockchains to create a global view of the participant's activity. Applications are, for instance, cross-chain tax audit [59],

or cross-chain portfolio tracking [60], and even cross-chain security, by representing and monitoring cross-chain state [37], all applications that could benefit from a more formal treatment that BUNGEE can provide.

In conclusion, BUNGEE offers three advantages compared to the related work: it is built on a theoretical basis that formally defines a blockchain view, the BVI framework; it provides a way to build participant-centric views composed of proofs that provide provenance-evidence; it defines algorithms for merging and processing views, allowing for a wide range of applications.

## 6. Conclusion

In this paper, we introduce the concept of blockchain view, a foundational concept for handling cross-chain state. Views represent different perspectives of blockchain participants, allowing one to reason about their different incentives and goals.

We present BUNGEE, a system that can create views from a set of states according to a projection function, yielding a collection of states accessible by a certain participant. BUNGEE can create a snapshot by retrieving the state of a blockchain, and based on participants' permissions, build a view of the global state. After that, BUNGEE creates extended states, the basis for merging blockchain views. Different views (possibly from different blockchains) can be merged into a consolidated view, enabling applications such as cross-chain audits and analytics. Finally, we discuss different aspects of BUNGEE, including decentralization, security, privacy, and its applications.

An important area for future work is on using zero-knowledge proofs to enhance view privacy. We would also like to empirically validate our work by providing an implementation of BUNGEE[4] that can provide support for building blockchain migrator applications.

## Acknowledgments

We warmly thank Luis Pedrosa, the colleagues of the academic paper workforce at Hyperledger, and the colleagues of IETF's ODAP working group for many fruitful discussions. We thank Afonso Marques, Benedikt Putz and Diogo Vaz for reviewing and providing valuable feedback on an earlier version of this paper. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and 2020.06837.BD, and the European Commission through contract 952226 (BIG). Rafael was supported by Blockdaemon.

## References

[1] J. Garay, A. Kiayias, N. Leonardos, The Bitcoin Backbone Protocol with Chains of Variable Difficulty, in: J. Katz, H. Shacham (Eds.), Advances in Cryptology – CRYPTO 2017, Springer International Publishing, Cham, 2017, pp. 291–323.

[2] R. Belchior, S. Guerreiro, A. Vasconcelos, M. Correia, A survey on business process view integration: past, present and future applications to blockchain, Business Process Management Journal 28 (3) (2022) 713–739, publisher: Emerald Publishing Limited. doi:10.1108/BPMJ-11-2020-0529.
URL https://doi.org/10.1108/BPMJ-11-2020-0529

[3] R. Dijkman, Diagnosing differences between business process models, in: International Conference on Business Process Management, 2008. doi:10.1007/978-3-540-85758-7_20.
URL https://link.springer.com/chapter/10.1007/978-3-540-85758-7_20

[4] M. Graf, D. Rausch, V. Ronge, C. Egger, R. Küsters, D. Schröder, A Security Framework for Distributed Ledgers, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, 2021, pp. 1043–1064, event-place: New York, NY, USA. doi:10.1145/3460120.3485362.
URL https://doi.org/10.1145/3460120.3485362

[5] M. Graf, R. Küsters, D. Rausch, Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric (2020). doi:10.1109/EuroSP48549.2020.00023.

[6] R. Belchior, L. Riley, T. Hardjono, A. Vasconcelos, M. Correia, Do you need a distributed ledger technology interoperability solution?, TechrxivPublisher: TechRxiv (Feb. 2022). doi:10.36227/techrxiv.18786527.v1.
URL https://www.techrxiv.org/articles/preprint/Do_You_Need_a_Distributed_Ledger_Technology_Interoperability_Solution_/18786527/1

[7] R. Belchior, L. Riley, T. Hardjono, A. Vasconcelos, M. Correia, Do you need a distributed ledger technology interoperability solution?, Distributed Ledger Technologies: Research and PracticeJust Accepted (Sep 2022). doi:10.1145/3564532.
URL https://doi.org/10.1145/3564532

[8] R. Belchior, P. Somogyvari, J. Pfannschmid, A. Vasconcelos, M. Correia, Hephaestus: Modelling, analysis, and performance evaluation of cross-chain transactions (Sep 2022). doi:10.36227/techrxiv.20718058.v1.
URL https://www.techrxiv.org/articles/preprint/Hephaestus_Modelling_Analysis_and_Performance_Evaluation_of_Cross-Chain_Transactions/20718058/1

[9] A. Lohachab, S. Garg, B. Kang, M. B. Amin, J. Lee, S. Chen, X. Xu, Towards Interconnected Blockchains: A Comprehensive Review of the Role of Interoperability among Disparate Blockchains, ACM Comput. Surv. 54 (7), place: New York, NY, USA Publisher: Association for Computing Machinery (Jul. 2021). doi:10.1145/3460287.
URL https://doi.org/10.1145/3460287

[10] R. Belchior, M. Correia, T. Hardjono, Gateway Crash Recovery Mechanism draft v1, Tech. rep., IETF (2021).
URL https://datatracker.ietf.org/doc/draft-belchior-gateway-recovery/

[11] R. Belchior, A. Vasconcelos, M. Correia, T. Hardjono, HERMES: Fault-Tolerant Middleware for Blockchain Interoperability, Future Generation Computer Systems (Mar. 2021). doi:10.36227/TECHRXIV.14120291.V1.

[12] T. Hardjono, A. Lipton, A. Pentland, Toward an Interoperability Architecture for Blockchain Autonomous Systems, IEEE Transactions on Engineering Management 67 (4) (2020) 1298–1309, conference Name: IEEE Transactions on Engineering Management. doi:10.1109/TEM.2019.2920154.

[13] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, T. Takeuchi, T. Kuhrt, R. Belchior, Hyperledger Cactus Whitepaper, Tech. rep., Hyperledger Foundation (2020).
URL https://github.com/hyperledger/cactus/blob/master/docs/whitepaper/whitepaper.md

[14] S. Abdullah, J. Arshad, M. Alsadi, Chain-net: An internet-inspired framework for interoperable blockchains, Distributed Ledger Technologies: Research and PracticeJust Accepted (Jun 2022). doi:10.1145/3554761.
URL https://doi.org/10.1145/3554761

[15] M. Lenzerini, Data integration: A theoretical perspective, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2002, pp. 233–246.

[4] https://rafaelapb.page.link/bungee-wip

[16] Y. Katsis, Y. Papakonstantinou, View-based Data Integration, in: Encyclopedia of Database Systems, Springer US, 2009, pp. 3332–3339. doi:10.1007/978-0-387-39940-9_1072.
URL https://link.springer.com/referenceworkentry/10.1007/978-0-387-39940-9_1072

[17] Hyperledger Foundation, Hyperledger Fabric Private Data (2020).
URL https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html

[18] N. Hewett, W. Lehmacher, Y. Wang, Inclusive deployment of blockchain for supply chains, World Economic Forum, 2019.

[19] S. Saberi, M. Kouhizadeh, J. Sarkis, L. Shen, Blockchain technology and its relationships to sustainable supply chain management, International Journal of Production Research 57 (7) (2019) 2117–2135. doi:10.1080/00207543.2018.1533261.
URL https://www.tandfonline.com/doi/full/10.1080/00207543.2018.1533261

[20] M. Bartoletti, S. Lande, L. Pompianu, A. Bracciali, A general framework for blockchain analytics, in: Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, 2017, pp. 1–6.

[21] L. Brünjes, M. J. Gabbay, Utxo-vs account-based smart contract blockchain programming paradigms, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2020, pp. 73–88.

[22] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al., Hyperledger fabric: a distributed operating system for permissioned blockchains, in: Proceedings of the thirteenth EuroSys conference, 2018, pp. 1–15.

[23] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008).
URL http://bitcoin.org/bitcoin.pdf

[24] D. Boneh, M. Drijvers, G. Neven, Compact multi-signatures for smaller blockchains, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2018, pp. 435–464.

[25] J. Benet, Ipfs-content addressed, versioned, p2p file system, arXiv preprint arXiv:1407.3561 (2014).

[26] E. Abebe, Y. Hu, A. Irvin, D. Karunamoorthy, V. Pandit, V. Ramakrishna, J. Yu, Verifiable Observation of Permissioned Ledgers, in: 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), IEEE, 2021, pp. 1–9.

[27] M. Al-Bassam, A. Sonnino, V. Buterin, Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities, arXiv preprint arXiv:1809.09044 160 (2018).

[28] P. Chatzigiannis, F. Baldimtsi, K. Chalkias, SoK: Blockchain Light Clients, Tech. Rep. 1657 (2021).
URL http://eprint.iacr.org/2021/1657

[29] P. Ruan, Y. Kanza, B. C. Ooi, D. Srivastava, Ledgerview: Access-control views on hyperledger fabric, in: Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 2218–2231. doi:10.1145/3514221.3526046.
URL https://doi.org/10.1145/3514221.3526046

[30] Y. Zhang, S. Wu, B. Jin, J. Du, A blockchain-based process provenance for cloud forensics, in: 2017 3rd IEEE international conference on computer and communications (ICCC), IEEE, 2017, pp. 2470–2473.

[31] A. K. Fedorov, E. O. Kiktenko, A. I. Lvovsky, Quantum computers put blockchain security at risk (2018).

[32] W. Buchanan, A. Woodward, Will quantum computers be the end of public key encryption?, Journal of Cyber Security Technology 1 (1) (2017) 1–22.

[33] T. M. Fernandez-Carames, P. Fraga-Lamas, Towards post-quantum blockchain: A review on blockchain cryptography resistant to quantum computing attacks, IEEE access 8 (2020) 21091–21116.

[34] R. A. Grimes, Cryptography apocalypse: preparing for the day when quantum computing breaks today's crypto, John Wiley & Sons, 2019.

[35] X. Yang, W. Li, A zero-knowledge-proof-based digital identity management scheme in blockchain, Computers & Security 99 (2020) 102050. doi:10.1016/j.cose.2020.102050.
URL https://www.sciencedirect.com/science/article/pii/S0167404820303230

[36] R. Belchior, B. Putz, G. Pernul, M. Correia, A. Vasconcelos, S. Guerreiro, SSIBAC : Self-Sovereign Identity Based Access Control, in: The 3rd International Workshop on Blockchain Systems and Applications, IEEE, 2020.

[37] I. Mihaiu, R. Belchior, S. Scuri, N. Nunes, A Framework to Evaluate Blockchain Interoperability Solutions, Tech. rep., TechRxiv (Dec. 2021). doi:10.36227/TECHRXIV.17093039.V2.
URL https://www.techrxiv.org/articles/preprint/A_Framework_to_Evaluate_Blockchain_Interoperability_Solutions/17093039

[38] EY, EY announces general availability of EY Blockchain Analyzer: Reconciler (2022).
URL shorturl.at/hpMQ8

[39] R. Belchior, A. Vasconcelos, S. Guerreiro, M. Correia, A Survey on Blockchain Interoperability: Past, Present, and Future Trends, ACM Computing Surveys 54 (8) (2021) 1–41.
URL http://arxiv.org/abs/2005.14282

[40] H. Bandara, X. Xu, I. Weber, Patterns for blockchain migration, arXiv preprint arXiv:1906.00239Publisher: Jun (2019).

[41] JP Morgan, Quorum White Paper (2017).
URL https://github.com/jpmorganchase/quorum/blob/master/docs/QuorumWhitepaperv0.2.pdf

[42] W. F. Silvano, R. Marcelino, Iota Tangle: A cryptocurrency to communicate Internet-of-Things data, Future Generation Computer Systems 112 (2020) 307–319. doi:10.1016/j.future.2020.05.047.
URL https://www.sciencedirect.com/science/article/pii/S0167739X19329048

[43] R3 Foundation, R3's Corda Documentation.
URL https://docs.r3.com/

[44] Hyperledger, Hyperledger Besu Ethereum client - Hyperledger Besu (2019).
URL https://besu.hyperledger.org/en/stable/

[45] T. Qiu, R. Zhang, Y. Gao, Ripple vs. swift: transforming cross border remittance using blockchain technology, Procedia computer science 147 (2019) 428–434.

[46] R. Henry, A. Herzberg, A. Kate, Blockchain access privacy: Challenges and directions, IEEE Security & Privacy 16 (4) (2018) 38–45.

[47] C. team, Introduction to Canton — Daml SDK 2.1.1 documentation (2021).
URL https://docs.daml.com/canton/about.html

[48] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, R. P. Liu, Survey: Sharding in Blockchains, IEEE Access 8 (2020) 14155–14181. doi:10.1109/ACCESS.2020.2965147.

[49] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, B. Ford, Omniledger: A secure, scale-out, decentralized ledger via sharding, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 583–598.

[50] M. Zamani, M. Movahedi, M. Raykova, Rapid-Chain: Scaling Blockchain via Full Sharding, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2018, p. 18, event-place: New York, NY, USA.
URL https://doi.org/10.1145/3243734.3243853

[51] A. Y. Levy, Logic-Based Techniques in Data Integration, in: Logic-Based Artificial Intelligence, Springer US, 2000, pp. 575–595. doi:10.1007/978-1-4615-1567-8_24.
URL https://link.springer.com/chapter/10.1007/978-1-4615-1567-8_24

[52] G. Verdian, P. Tasca, C. Paterson, G. Mondelli, Quant overledger whitepaper, Release V0. 1 (alpha) (2018).

[53] Blockdaemon, Ubiquity.
URL https://blockdaemon.com/platform/ubiquity/

[54] G. Wood, Polkadot: Vision for a heterogeneous multi-chain framework, White Paper 21 (2016) 2327–4662.

[55] Polkadot, Cross-Consensus Message Format (XCM) · Polkadot Wiki (2021).
URL https://wiki.polkadot.network/docs/learn-crosschain

[56] J. Kwon, E. Buchman, Cosmos whitepaper, A Netw. Distrib. Ledgers (2019).

[57] A. M. Rozario, C. Thomas, Reengineering the audit with blockchain and smart contracts, Journal of emerging technologies in accounting 16 (1) (2019) 21–35, publisher: American Accounting Association.

[58] Y. Jo, J. Ma, C. Park, Toward Trustworthy Blockchain-as-a-Service with Auditing, in: ICDCS, 2020. doi:10.1109/ICDCS47774.2020.00068.

[59] A. Li, G. Tian, M. Miao, J. Gong, Blockchain-based cross-user data

shared auditing, Connection Science (2021) 1–21Publisher: Taylor & Francis.

[60] Blockdaemon, Introducing Blockdaemon's New Staking Dashboard (Sep. 2021).
URL https://blockdaemon.com/blog/introducing-blockdaemons-new-staking-dashboard/