Is My Perspective Better Than Yours? Blockchain Interoperability with Views

Rafael Belchior, Limaris Torres, Jonas Pfannschmidt, André Vasconcelos, Miguel Correia

Abstract

Distributed ledger technology (DLT) provides decentralized and tamper-resistant data storage, replicated among mutually untrusting participants. With the advancement of this technology, different privacy-preserving blockchains have been proposed, such as Corda, Hyperledger Fabric, and Digital Asset's Canton. These distributed ledgers only provide *partial consistency*, which implies that participants can view the same ledger differently. A *view* represents the states of a blockchain available to a particular stakeholder. The combination of views forms an integrated view that represents a consistent global state shared by all participants.

This paper introduces BUNGEE (Blockchain UNifier view GEnErator), the first DLT view generator, to allow capturing DLT snapshots, constructing views, and performing arbitrary operations on those, such as integrating views. Creating and integrating views allows interesting applications, such as stakeholder-centric snapshots for audits [21], cross-chain analysis [42], blockchain migration, and data analytics [5].

1 Introduction

Blockchains[1] provide trustworthy and transparent services, leveraging a network of mutually untrusting participants. A highly desirable property of DLTs is consistency [22]. Consistency states that all honest parties share a common prefix of the blockchain. Based on this assumption, each ledger holds a single source of truth for all its *participants*[2]. Consistency is typically the foundation

---

[1] We use the terms DLT and blockchain interchangeably. A DLT subsumes a blockchain, i.e., a blockchain is a DLT.

[2] A similar assumption could be extended to the business process research area, 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 [8]. To account for the multitude of business process views, *business view process integration* (BVPI) studies the consolidation of different views regarding a business process [20, 8]. Business view

of the decentralized trust that DLTs offer.

However, some permissioned DLTs offer *partial consistency* instead of (full) consistency, making a trade-off between transparency and privacy. Informally speaking, partial consistency means that "parties may see only part of the state, but these views put together should result in a consistent global state" [24]. Blockchains providing partial consistency have been putting resources on enabling interoperability with other blockchains, following the growing tendency of the space to accommodate DLTs offering different properties and features [10].

A problem naturally emerges from a multi-chain ecosystem: since participants might have different views of a chain, 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 to do so if that blockchain is private, or if the blockchain provides partial consistency?

We believe that the 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 state from different blockchains in a standardized way. This way, views help an external observer to reason about partial consistency in DLTs. On the other hand, in public DLTs with probabilistic consensus, conflicting (usually) temporary views on the same ledger exist (i.e., forks) but are resolved by a rule (e.g., longest chain rule), and thus consistency holds.

In this context, building and analyzing views is important to accurately understand each stakeholder's view of each DLT at every moment (including in public blockchains [43]) as a tool for business intelligence (e.g., understanding better a certain protocol) and auditing (e.g., monitoring a protocol). Views directly support blockchain interoperability since it would now be easier to share the perspectives of all participants across heterogeneous DLTs [5, 41], allowing for 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, 7, 12, 25].

In this paper, we propose 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 taking into account the desired time interval. Furthermore, we provide support for views to be merged in the merging phase. Merging views is a data integration process that follows the *Global as View* approach [38]. After different view generators create partial views, a merge operation may be applied to the views to produce a consolidated view, according to a *merging algorithm*. Since BUNGEE can integrate views, it is considered a view integration system [37, 8, 34]. We focus our contributions around two research questions:

---

process integration (BPVI) addresses the challenges of processes involving several participants with different incentives, alleviating them by merging models that represent a different view of the same model.

Research Question 1. *How to generate blockchain views?*

There are multiple DLT data formats, a consequence of their architecture, consensus, and even identity models. A formalization of the blockchain view and surrounding concepts is necessary to reason about data representation across chains.

We first present the Blockchain View Integration Framework (BVIF) a collection of concepts surrounding the view, rooted in the state abstraction and causality relationships between transactions, states, and views. BVIF is the foundation to formally describe the algorithms that BUNGEE uses 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.

Research Question 2. *How can one merge views and create an integrated view?*

Creating views allows one to see one stakeholder's perspective over the whole network. However, how to obtain a holistic view of a DLT providing partial consistency, i.e., the perspectives of all participants combined? We ensure that the view's creation, merging, and processing come with integrity and accountability guarantees. As a core contribute of this paper, we formally describe the algorithms for each step to merge and process a view, providing a comprehensive discussion about decentralization, efficiency, and privacy trade-offs.

Paper Outline

This document is organized as follows: Section 2 introduces the background necessary to comprehend this paper. In Section 3 we introduce the blockchain view integration problem and propose the Blockchain View Integration Framework (BVIF), a collection of concepts around BUNGEE. Next, Section 4 presents BUNGEE, including the system model (Section 4.1), the snapshotting phase (Section 4.2), the view building phase (Section 4.3), the view processing phase (Section 4.4), and the discussion (Section 4.5). After that we present the related work, in Section 5. Finally, we conclude the paper (Section 6).

2 Preliminaries

This section presents the necessary background to comprehend the paper, along with motivating use cases.

Blockchains Providing Partial Consistency

Blockchains providing partial consistency are blockchains that partition the global state according to some criteria, implying that there will be different views on the global state. Private blockchains require their participants to be authenticated and only expose their content to trusted parties (although those parties do not necessarily trust each other). In private blockchains, different views are not only common, but desirable for privacy reasons [23]. Parties may want to share information with a selected group.

For example, Hyperledger Fabric (Fabric), a private blockchain framework, provides a feature called private data collections. Private data allows sets of participants to hide part of the state they hold, only sharing a hash of that private data as proof of existence [3, 29]. This feature effectively implements partial consistency in Fabric, allowing for the existence of different views. In Corda [47], transactions are ordered as a set of (potentially) disconnected directed acyclic graphs – parties can access certain sub-graphs, i.e., Corda provides partial consistency. Other examples exist, such as Quorum [33], IOTA [52] and Digital Asset's Canton [53]. A visualization of the concept is shown in Figure 1.

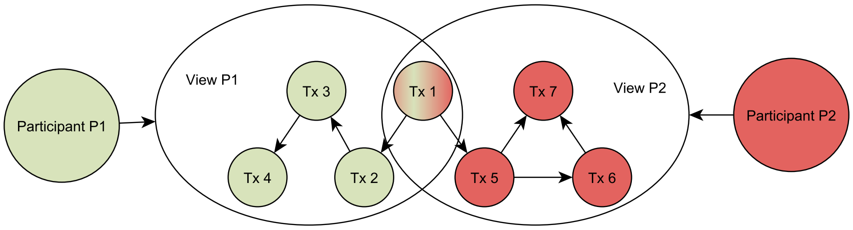

Figure 1: Two different participant views over the same DLT. Tx stands for transaction. A green or red labeled transaction is available (read access) to Participant P1 or Participant P2, respectively. Transaction Tx1 is available for both participants.

Blockchain Interoperability

The emergence of many blockchains raised the debate on the need for interoperability amongst them. Interoperability can be defined as the ability of multiple parties to work together by sharing/exchanging information [55].

A recent survey classifies blockchain interoperability solutions into three categories: cryptocurrency-directed approaches, blockchain engines, and blockchain connectors [13]. *Public connectors* are interoperability solutions allowing for public blockchains to connect. *Blockchain of blockchains* are frameworks allowing for the reuse of infrastructure that powers *application-specific blockchains*: blockchains powering a specific decentralized application. In those, the users can configure their instance of a blockchain, customizing its behavior to their application needs (e.g., consensus algorithm, governance model, and reward model). Blockchains created with those frameworks interoperate with each other. Finally, *blockchain connectors* connect public to private blockchains and centralized systems. Solutions in this category comprise centralized solutions (trusted relays), general solutions that require blockchain refactoring (blockchain-agnostic protocols), and blockchain migrators. Other surveys also cover this research area [41, 18, 60, 48].

Use cases benefiting from blockchain views

The first use case is cross-chain state creation, managing, and visualization. While some introductory work has been made [42, 5], it is hard 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 enhance view analysis for auditors, cybersecurity experts, and developers. Auditors and cybersecurity professionals can facilitate audits because different data partitions can be analyzed from a specific angle. Developers can have insights into their applications and processes. This applies to public blockchains as well. The representation of on-chain data through a DLT view in multiple chains allows for a visualization of cross-chain state, making it easier to manage and reason about it. A specific application could be having one view across multiple Cosmos zones, Polkadot parachains, or across Layer 2 solutions (Polygon, Arbitrum, and others, for instance) [13].

The second use case is decentralized application migration. Migration of blockchain-based applications is not only necessary, but increasingly common [4, 13, 8]. Migration allows enterprises to experiment with different 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 to move it to a different DLT infrastructure. With several views on participants' concerns operating on the source blockchain, one might need to consolidate their diverse views into an integrated view, one that serves as the foundation of the migration. The integrated view comprises a holistic view of the application's state at the source DLT. That view can then be transferred to the target infrastructure, along with its functionality (i.e., smart contract migration). We leave the treatment of this interesting problem and its details (for example, how to manage user keys) for future work.

Finally, the third use case is allowing cross-chain asset transfers [11, 12, 6]. Transferring assets between public DLTs and centralized systems (or private DLTs) is hard because it relies on strong trust assumptions or transparency. An asset transfer is typically implemented by locking an asset on the source chain and unlocking it on the target chain. However, if one of the chains is private (or centralized), such a state is not visible (by design). Therefore, decentralized transfers across these system types rely on proofs (or, rather, notarizations) of the current state of each chain concerning the representation of an asset. Blockchain views can be the bridge enabling decentralized blockchain interoperability across heterogeneous systems, by representing such notarizations on a public forum, with a standardized data format, independent of any specific blockchain implementation. A related use case to this would be to build a cross-chain wallet that, giving a private key, outputs all tokens in all blockchains associated with that wallet. This could be particularly useful for people that had LUNA tokens spread across several blockchains, especially after the coin plummeted in value [31].

3 Blockchain View Integration Framework

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

3.1 Motivation Case Study

To illustrate how such problems can provide a solution, we present a typical use case on private blockchains, supply chain [30], that benefits from representing the various internal views to an external observer.

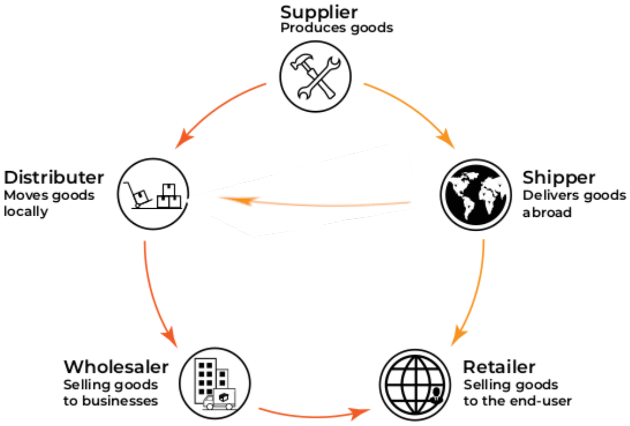

Figure 2: Supply chain scenario with five participants engaging in asset trading.

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, enterprises 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 needed for auditability and typically can be tampered with, leading to the suitability of blockchain to address these problems by monitoring the execution of the collaborative process, ensuring that the process execution is in compliance with business rules [27, 51]. Audits inspect the trail of transactions referring to a product's lifecycle, so 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? 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 enrolling 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 channel that produce, transport, and trade, as illustrated by Figure 2:

- A Supplier, producing goods.
- A Shipper, moving goods between parties.
- A Distributor, moving goods abroad. Buys goods from Suppliers and sells them to Wholesalers and Retailers.
- 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 also want to have a different relationship with the Wholesaler. It charges them a lower price than it does with the Retailer (as it sells assets in bulk). Finally, the Wholesaler may also want to have a private relationship with the Retailer and the Shipper (as it charges them a higher price than the Wholesaler). These private relationships are the *domains* of our use case.

Domains hold a subset of the ledger only accessible by authorized parties. By sectioning the shared ledger, different views over 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 1 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 to access to its price's hash 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 3, translate into three different price values for the same item – five participants originate five different views.

Let us now suppose 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. The different domains are illustrated by Figure 3.

All domains together represent the whole ledger. All views of the users allow to compose the integrated DLT view $\mathcal{V}$.In d_1, the Distributor, Supplier, and Shipper share information regarding the asset ID and its respective price, which is the lowest price across views. Note that the Distributor has access to d_1 and d_3, the Shipper can access d_1 and d_2, and the Wholesaler can access d_2 and d_3. As every stakeholder has a different combination of the domains that are

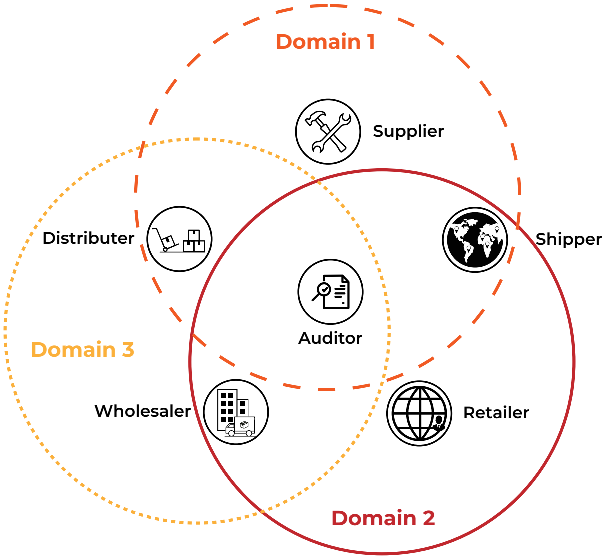

Figure 3: Different domains on the blockchain supporting the supply chain scenario. For instance, the Supplier has access to domain 1, while the Shipper acesses domains 1 and 2. Access to different domains leads to the creation of different views.

accessible, the DLT infrastructure yields five different views. The unique combination of each stakeholder's participation in the different domains originates a view, as illustrated in Table 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; a third option would be to hide all prices. In particular, the different views are translated to 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 managing the blockchain, and thus several options are possible. We will address this point later in this paper.

3.2 Blockchain View Integration Framework

In this section, we formally define the necessary terms for blockchain view integration, the *Blockchain View Integration Framework* (BVIF). BVIF provides

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

Participant\Domain	d_1	d_2	d_3	
Supplier	ID: 1 Price: 1	ID: 1 Price: hidden	ID: 1 Price: hidden	v_1
Shipper	ID: 1 Price: 1	ID: 1 Price: 2	ID: 1 Price: hidden	v_2
Distributor	ID: 1 Price: 1	ID: 1 Price: hidden	ID: 1 Price: 3	v_3
Wholesaler	ID: 1 Price: hidden	ID: 1 Price: 2	ID: 1 Price: 3	v_4
Retailer	ID: 1 Price: hidden	ID: 1 Price: 2	ID: 1 Price: hidden	v_5

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 implementing 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. `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` $\rightarrow \{v, \emptyset, \perp\}$. `store` 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 UXTO blockchains (e.g., Bitcoin) or more complex ones, an account model (e.g., Ethereum), or others by combining the operations mentioned above (e.g., Hyperledger Fabric).

- $\mathcal{A}$ 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 [44] as follows: the database (collection of all states) is a list of UTXO entries (states). A UTXO has a unique identifier, the transaction hash, 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 the current input refers to. 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 straighforward 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 write 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 veryfing key.
- id is the unique identifier of the participant. It is the output of a function over the participant's public key. For instance, Bitcoin addresses are used to uniquely identify Bitcoin accounts and are formed by double-hashing the public key associated with that account.

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 transact on DLTs (by redirecting signed transactions to them). In practice, nodes mediate read and write operations issued by participants. We introduce the concept of Access Point to formalize the relationship between participants and nodes as follows:

Definition 3. *Access Point (AP).* An AP ω 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, ω^{-1} returns the node set n_p that a participant can access, i.e., $\omega^{-1}(\upsilon) \rightarrow n_p$.

Nodes can access a DLT via a primitive `obtainDLT`. The result of $\texttt{obtainDLT}(n_p) = \mathcal{B}$. An access point tells us which participants can access the ledger via a specific node. The information that participants can read and write in the ledgers through nodes falls into the concept of *virtual ledger*:

Definition 4. *Virtual ledgers.* A virtual ledger $\mathcal{L}_v$ is a projection of a ledger $\mathcal{L}$ 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_π of the database 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 (or $\emptyset$, if the participant is not authorized to access the ledger). This corresponds to "what the participant can see".

$d_{\pi_{l,p_n}}$	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 3.1. 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.

Let us recall that the database or a subset of it 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. Let us consider the following projections, illustrated by Figure 2:

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

We define a function `obtainVirtualLedger` that receives as input a ledger $\mathcal{L}$ and a projection function $\mathcal{F}_p$ and returns a projection $d_{\mathcal{L},\mathcal{F}_p}$ or the empty set. Each projection from a virtual ledger maps a participant to states. Some projections are not unique (e.g., $d_{\mathcal{L},\mathcal{F}_{p_5}}$ and $d_{\mathcal{L},\mathcal{F}_{p_6}}$). This concept is the basis of the DLT view, which we will define later in this section.

Definition 5. *DLT System.* Let $\mathcal{T}$ be a set of ordered transactions, $\mathcal{L}$ the ledger, $\mathcal{N}$ the set of nodes, Υ the set of participants, and τ a global clock. A DLT is a tuple $\mathcal{B} \doteq (\mathcal{L}, \mathcal{W}, Validate, Consensus, \tau)$, where:

- Υ is the set of participants, the entities accessing and interacting with the blockchain. Each participant $p \in \Upsilon$ can interact with the blockchain via a node $n \in \mathcal{N}$.

- A set of ordered transactions $\mathcal{T}$.
- The transaction pool $\mathcal{W}$ which holds the transactions to be included in $\mathcal{L}$.
- $Consensus$, a predicate on $\mathcal{W}$ and on consensus information from a subset of nodes $\in \mathcal{N}$ and returns true or false: $Consensus(\mathcal{W}, \mathcal{N}) \implies \{true, false\}$. Consensus defines the rules for valid state transitions.
- $Validate$, an algorithm taking as input a ledger and the pool of transactions, and yielding an updated ledger that incorporates a subset of the transactions from the transaction pool: $(\mathcal{L}', \mathcal{W}') \overset{Consensus}{\longleftarrow} (\mathcal{L}, \mathcal{W})$, with $\mathcal{L}' = \mathcal{L} \cup \mathcal{W} \setminus \mathcal{W}'$. The application of $Validate$ is only successful upon the $Consensus$ predicate over $\mathcal{W}$.
- The clock, τ, captures the order of transactions, mapping ledger rounds r to a time t, i.e., $\tau : s \longrightarrow t$. The clock can output the time from transactions and states since it retrieves the round associated with each.

Both ledgers and virtual ledgers that support DLTs 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 6. *Transaction.* A transaction t is a tuple $(tid, t_i, target_i, payload_i, \sigma_{K_s^p}(tid, t_i, target_i, payload_i), S_t$ where:

- tid is a unique increasing sequence identifier for a transaction. This identifier allows 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_i is the transaction timestamp
- $target_i$ is the target of the transaction (corresponding to a state).
- $payload_i$ is the transaction payload. The payload can carry arbitrary information (smart contract parameters, UTXO input value).
- a signature on the transaction $\sigma_{K_s^p}(tid, t_i, target_i, payload_i)$, where $s_{k,v}$ from the creator of the transaction, p.
- S_{tid}, a set of input states given as input to transaction with transaction ID sid.

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 ρ, 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 7. *State.* A state s is a tuple $(s_k, s_{k,v}, t, \pi_k)$, where

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

$$s_{k,0} \xrightarrow{t_1} s_{k,1} \xrightarrow{t_2} s_{k,2} \xrightarrow{t_{k-1}} s_{k,v}$$

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). 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.

We define the *cardinality* of a state s_n as $|s_n|$ as the number of transactions composing it. If s_n has a set of transctions $T = \{t_1, ..., t_i\}$, then $|s_n| = i$. The state of a particular object can be reconstructed from the execution of all the transactions that refer it. The global state is then the set of all states, the set $\mathcal{S}$.

The set of states visible by a certain participant (states that authorize the participant to read/write/update) is what we call a view of the blockchain. A view is an abstraction at a certain point of time that encompasses the set of states a participant can access.

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

Definition 8. *View.* A view $v_{\mathcal{L}_v,p}$ is a projection of a virtual ledger $\mathcal{L}_v$, in the form of $(v_k, v_{k_{time}}, p, d_{\pi_{l,p}}, S_{v_{k,v}}, \Pi)$, where

- its key, v_k, is a unique ID
- the projection function $d_{\pi_{l,p}}$ that originated the view
- an initial time t_i, and a final time t_f, that constitute the time interval $t_k = t_f - t_i$. The time interval restricts the states belonging to that view to the specified time limit. 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 participant *upsilon* can be associated with a node n, accessible by a blockchain access point ω.
- $S_{v_{k,v}}$ corresponds to the set of versioned states held by v.
- Π is a set of proofs accounting for the validity of a view (e.g., accumulator value for over states ordered by last update).

A DLT view corresponds to the sets of different values referring to the same key that a participant can access.

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 $\mathcal{L}$.

Definition 9. *View Cardinality.* Let there be a DLT view $v_{l,p}$ belonging to a participant p. A view $v_{l,p}$ has cardinality i when the the number of states composing that view is i, i.e., $d_{\pi_{l,p}} = \{s_1, ..., s_i\}$. In other words, $|v_{l,p}| = i$.

Definition 10. *DLT Domain t* is a tuple $(d, s_k, s_{k,v}, \mathcal{F}_d, P, value)$, where:

- d is the identifier of the domain
- s_k is the state key which that domain referrers to.
- $s_{k,v}$ is the state value corresponding to s_k.
- projection function that generates the state value $s_{k,v}$ of domain d indexed by s_k.
- P is the set of participants that can read the value v of state s_k on domain d.

Domains represent private relationships; they capture how many participants can share the same state. Domains then capture if a state is accessible (value readable) or not by a set of participants. The same state's key can have different values on different domains (depending on the projection function generating the domain). Participants sharing the same domain do not imply having the same view (views are not necessarily equivalent, see Definition 11).

Definition 11. *View equivalence.* Let there be a ledger $\mathcal{L}$ composed by a set of views $\{v_1, v_2, ..., v_n\} \in \mathcal{V}$, holding respectively the sets of states $\{S_1, S_2, ..., S_n\} \in \mathcal{S}$, i.e., there are the pairs $\{(v_1, S_1), (v_2, S_2), ...(v_n, S_n)\}$. There is view equivalence, denoted by $\texttt{equivalent}(v_i, v_j)$ if, for any pair of views $v_i, v_j \in \mathbb{V}$ there is a bijection $\varphi : s_i \to s_j$ such that if both sets of states from the views are the same, their views are equivalent, i.e., $\varphi(s_i) = s_j \implies \forall s \in s_i, s$ could replace all $s' \in s_j \implies v_i \equiv v_j$.

Following the example of Table 2, v_1 and v_6 are equivalent views because the states that are accessible to those views are the same. However, those views are not equal, as the other parameters might change.

A view can be unique:

Definition 12. *View Uniqueness.* Let there be a ledger $\mathcal{L}$ originating a set of views $\{v_1, v_2, ..., v_n\} \in \mathcal{V}$, constructed by sets of states $\{S_1, S_2, ..., S_n\} \in \mathcal{S}$, in which views are associated with their set of states $\{(v_1, S_1), (v_2, S_2), ...(v_n, S_n)\}$. Let us call each pair $(v_n, S_n) = \epsilon_n$. Let $\mathbb{V}$ be a monotone collection of non-empty subsets of $\mathcal{V}$, i.e., $\mathbb{V} \subseteq 2^{\{v_i, v_j, ..., v_n\}} \setminus \emptyset$. A view v_i is unique, denoted by $\texttt{unique}(v_i)$, if, for any pair V_i, V_j of monotone $\mathbb{V}$ there is NOT a bijection $\varphi : s_i \to s_n$ such that $\forall s_i \in \epsilon_i, s_i = s_j$, with $s_j \in \epsilon_j$.

Definition 13. *View Transparency.* Let there be a ledger $\mathcal{L}$ with a set of participants Υ and a set of DLT views $\{v_1, ..., v_n\} = \mathcal{V}$. We define the transparency grade κ of a DLT view v_v, denoted as $\kappa(v_v)$, as the ratio of participants who can access the states encoded in that view. More formally,

$$\kappa(v_v) = \frac{\sum_{i \neq v}^{n} \forall v_i [\texttt{equivalent}(v_n, v_j) + 1]}{|\mathcal{V}|}$$

This concept is useful to understand how many participants can access a certain set of states. Taking example from Table 2, $\kappa(v_{l,\mathcal{F}_{p6}}) = \frac{2}{6}$ because the view created by projecting ledger l with $d_{\mathcal{F}_{p6}}$ is equivalent with v_5 (summing with the view being compared). Therefore, two out of six participants can access the same set of states (this is, each participant has a different view, apart from p_5 and p_6).

4 BUNGEE, a Multi-Purpose View Generator

In this section, we present BUNGEE. First, we present the system model, key management processes, and adversary model. After that, we present the snapshot process. Next, we present how are views built, and then merged. Finally, we discuss the processes around creating snapshots, views, and merging views.

4.1 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 Υ: 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 not valid.

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, meaning that they follow the DLT protocol and establish consensus with other honest nodes – as a consequence, they report 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). Finally, nodes can be malicious but cautious, meaning that nodes are only malicious if there are no accountability checks that can penalize them (this is if nodes know that there 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 the snapshot. After the snapshot is captured, it is redirected to a controller, who can create, process, and distribute views.

To achieve the proposed goals, BUNGEE leverages a three-phase process, following our definition of view generator (see Figure 4).

Firstly, each state accessible by each stakeholder is collected in the snapshotting phase. States are processed, and a representation of the ledger that the participant has access to is built (this is, the state is collected using a specific projection function). An example of a projection function behavior could be retrieving all the states (versions included) and their respective values from a participant's perspective (according to the domain the participant can access).

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 that one can see. 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. Proofs of the view can be published in a public forum.

Finally, the view merging phase, which is optional, comprises merging views into an integrated one. Different views obtained from other generators require communication with other BUNGEE instances (the diagram shows a simplification. After that, an extended state is created from the states present on each view that share the same key. Finally, a merging algorithm is applied over the extended state. Finally, each view generator signs the integrated view, which can optionally be published in a public forum.

Once the three steps are completed, the views (the generated view and optionally the integrated view are returned by BUNGEE to a client application (for example, a blockchain migration application). Due to the modularity that BUNGEE offers, adding support for different applications is facilitated. Next, we present each phase depicted in this overview in finer detail.

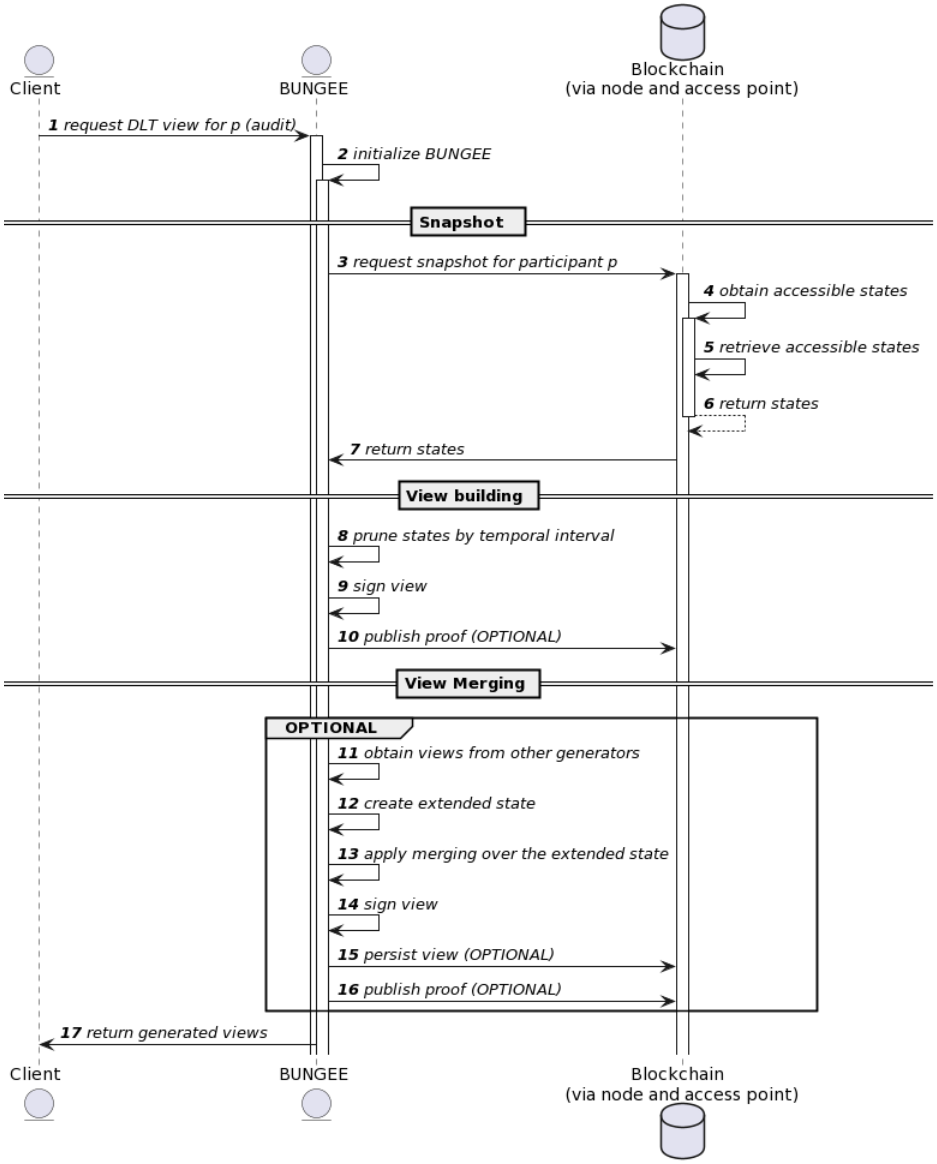
Client
BUNGEE
Blockchain
(via node and access point)
1 request DLT view for p (audit)
2 initialize BUNGEE
Snapshot
3 request snapshot for participant p
4 obtain accessible states
5 retrieve accessible states
6 return states
7 return states
View building
8 prune states by temporal interval
9 sign view
10 publish proof (OPTIONAL)
View Merging
OPTIONAL
11 obtain views from other generators
12 create extended state
13 apply merging over the extended state
14 sign view
15 persist view (OPTIONAL)
16 publish proof (OPTIONAL)
17 return generated views
Client
BUNGEE
Blockchain
(via node and access point)

Figure 4: Sequence diagram detailing the interactions between a BUNGEE client application, an instance of BUNGEE, and a blockchain.

4.2 Snapshot

A snapshot is a set of states that a certain stakeholder can access, plus proofs 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 state 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., `snapshot` $\doteq \{id, v, sb, t_i, t_f, T\}$. 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. Versioning snapshots allows for efficiently building snapshots from older snapshots (this is, building snapshots from incremental changes from older snapshots), similarly to how Git tracks updates to the files it manages.

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 re-execute all transactions. For each transaction, a BUNGEE has to check its target. If there is no state key with a target equal to the current transaction, BUNGEE creates a new state. 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).

Building 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 27 and 28), and, finally, the algorithm returns a snapshot.

4.3 View Building

This section explains how views are built, therefore answering 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 boundaries (line 7). If there are, each state in the snapshot is included if it belongs to the temporal limit (line 17) and removed otherwise (line 14). Finally, the view generator signs the view (line 19) and returns it to the client application (line 20).

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 $n$, snapshot
snapshot.id ← $\texttt{snapshot}_{id}$
snapshot.v ← $1$
snapshot.sb ← $\emptyset$
snapshot.$t_i$ ← $\bot$
snapshot.$t_f$ ← $\bot$
$t_{it} \leftarrow \infty$    ▷ temporary variable to hold minimum state timestamp to date
$t_{ft} \leftarrow 0$ ▷ temporary variable to hold maximum state timestamp to date

$n = \omega^{-1}(p)$    ▷ choose any available node
$\mathcal{B} =$ obtainDLT($n$)    ▷ depends on the DLT client implementation
$\mathcal{L} = \mathcal{B}.\mathcal{L}$
$d_{\mathcal{L},\mathcal{F}_p} =$ obtainVirtualLedger($\mathcal{L}, \mathcal{F}_p$)    ▷ obtain projection of $\mathcal{L}$ according to $p$
foreach $s \in d_{\mathcal{L},\mathcal{F}_p}$ do
    $s_{k,it} \leftarrow \emptyset$ ▷ the timestamp of the first transaction applied to state $s_k$
    $s_{k,lt} \leftarrow \emptyset$ ▷ the timestamp of the last transaction applied to state $s_k$
    snapshot.$sb[s_k].s_k = s_k$
    snapshot.$sb[s_k].version = d_{\mathcal{L},\mathcal{F}_p}[s_k].T.length$
    snapshot.$sb[s_k]$.latestValue $= d_{\mathcal{L},\mathcal{F}_p}[s_k].s_{k,v}$
    snapshot.$sb[s_k].T = d_{\mathcal{L},\mathcal{F}_p}[s_k].T$    ▷ save list of transactions referring to each state key
    $s_{k,it} = d_{\mathcal{L},\mathcal{F}_p}[s_k].T[0]$    ▷ transaction list is ordered chronologically
    $s_{k,lt} = d_{\mathcal{L},\mathcal{F}_p}[s_k].T.length$
    if $s_{k,it} < t_{it}$ then
        $t_{it} = s_{k,it}$    ▷ update the auxiliary first timestamp
    end if
    if $s_{k,lt} > t_{ft}$ then
        $t_{ft} = s_{k,lt}$    ▷ update the auxiliary last timestamp
    end if
end foreach
snapshot.$t_i = t_{it}$
snapshot.$t_f = t_{ft}$
return snapshot
```

4.4 Merging views

In this section, we describe how to merge views and answer the research question *How can one merge views, and create an integrated view?*. The merging of views

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

```
Input: Snapshot snapshot, initial time $t_i$, final time $t_f$
Output: View $\mathcal{V}$
$\mathcal{V}.t_i \leftarrow t_i$
$\mathcal{V}.t_f \leftarrow t_f$
$\mathcal{V}.d_{\pi_{l,p}} \leftarrow \texttt{snapshot}.\mathcal{F}_p$
$\mathcal{V}.p \leftarrow \texttt{snapshot}.p$
$\mathcal{V}.\Pi \leftarrow \perp$
$\mathcal{V}.S_{k,v} \leftarrow \perp$
if $t_i <$ snapshot.$t_f$ OR $t_f >$ snapshot.$t_i$ then
    return;   ▷ there are no intersecting states that we want to capture, on the snapshot
end if
                                        ▷ each $sb = \{s_k, s_{k,\overrightarrow{v}}, \mathrm{v}\}$
foreach $s_k \in$ snapshot.$sb$ do
    foreach $t \in s_k$ do
        if $t.timestamp < t_i$ OR $t.timestamp > t.f$ then
            $\texttt{snapshot}.sb[s_k] \leftarrow \texttt{snapshot}.sb[s_k].T \setminus t$   ▷ removes transaction that is not within the specified time frame
        end if
    end foreach
    $\mathcal{V}.S_{k,v} \leftarrow \texttt{snapshot}.sb[s_k]$
end foreach
$\mathcal{V}.\Pi \leftarrow sign_{\mathbb{G}}(\mathcal{V})$
return $\mathcal{V}$
```

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 compare 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 14. 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 T;
- a proof of of state validity;
- metadata, which holds a list of operations that have been applied to the extended state;
- 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.,

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

After we initialize the list of extended states, 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 for each view to be merged. 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 14).

On line 18, 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 20 and 21) 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 have to sign $\mathcal{I}$ (line 21) to promote accountability. Signing the integrated view can be done in a distributed way using a multi-signature algorithm (for example, BLS Multi-Signatures [17]).

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. If there is no value for the first view, that state is removed. 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 certain view.

  Algorithm 5 prunes the values belonging to a certain 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.

4.5 Discussion

In this section, we discuss BUNGEE. The proliferation of blockchain interoperability solutions will create an exponential interest in exploring cross-chain logic and the need to model and analyze it. Our proposal constitutes the foundation to make sense of that diversity by allowing us to create views and integrate views from different blockchains systematically.

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 $\mathcal{I}$
$\mathcal{S} \leftarrow []$  ▷ 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
$\mathcal{I}.t_i \leftarrow \emptyset$
$\mathcal{I}.t_f \leftarrow \emptyset$
$\mathcal{I}.d_{\pi_{l,p}} \leftarrow \bigcup_{i=0}^{n} \mathcal{V}_n.d_{\pi_{l,p}}$
$\mathcal{I}.p \leftarrow \bigcup_{i=0}^{n} \mathcal{V}_n.p$
$\mathcal{I}.\Pi \leftarrow \perp$
$\mathcal{I}.S_{k,v} \leftarrow \perp$
foreach $v \in \mathcal{V}$ do
    foreach $s \in v.S_{k,v}$ do
        if $s \in \mathcal{S}$ then
            $\mathcal{S}[\overrightarrow{s.k}] = \mathcal{S}[\overrightarrow{s.k}] \cup \overrightarrow{s_{k,v}}$  ▷ if state exists, add value referring to that state, from current view
            $\mathcal{S}[\overrightarrow{s.k}].version \leftarrow \mathcal{S}[\overrightarrow{s.k}].version + 1$
        end if
        else
            $\mathcal{S}[\overrightarrow{s.k}] = \overrightarrow{s_{k,v}}$  ▷ otherwise, initialize state key list
            $\mathcal{S}[\overrightarrow{s.k}].version \leftarrow 0$
            $\mathcal{S}[\overrightarrow{s.k}].metadata \leftarrow \{MERGE - INIT\}$
        end if
    end foreach
end foreach
$\mathcal{I}.S_{k,v} = \mathtt{call}_{\mathtt{algorithm}\mathcal{M}}(\mathcal{S})$  ▷ OPTIONAL. Computes the state list of the integrated view according to $\mathcal{M}$ (see for example algorithm 5)
$\mathcal{I}.d_{\pi_{l,p}} \leftarrow \mathcal{I}.d_{\pi_{l,p}} \cup \{\mathcal{M}\}$  ▷ add reference to the merging algorithm
$\mathcal{I}.t_i = \mathtt{min}\{\mathcal{I}.S_{k,v}.t_i\}$  ▷ initial timestamp correspond to the initial timestamp of the processed states
$\mathcal{I}.t_f = \mathtt{min}\{\mathcal{I}.S_{k,v}.t_f\}$
$\mathcal{I}.\Pi \leftarrow sign_{\mathbb{G}}(\mathcal{I})$  ▷ signed collectively by $\mathbb{G}$
return $\mathcal{I}$
```

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 $\mathcal{S}'$
```

Do views provide integrity and accountability guarantees?

The existence of proofs on states are proofs of creation by the entity or set of entities that executed the transactions referring to that state. For example, a signed transaction hash qualifies as a 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 it, merges views, applies a merging algorithm, or notarizes the view as truthful. This set of proofs allows independent parties to validate the truthfulness of the view (by verifying each state) and hold view generators accountable for them.

The existence of the metadata fields, both on each extended state and on the views that encompass, allows 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 [14] (a distributed peer-to-peer file system maintained by a network of public nodes), or a public blockchain, similarly to some related work [12, 1]. In case a view is deemed to be false, automatic view conflict detection and resolution can take place.

An honest view generator, connected to an honest internal node, holds the knowledge of the view v, and can publish 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 by 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 publicised 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 is colluding (i.e., blockchain with financial information). If all internal nodes collude, the ledger is unreliable 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. We leave a more exhaustive security analysis for future work, when we can analyse the impact of several attacks, such as Eclipse attacks, where the attacker isolates a blockchain node by connecting several of their nodes to the victim [26]. A participant connecting to a sufficiently big number of nodes should be able to alleviate such attack.

Can one prove facts with blockchain 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. [1] sheds some light on this matter, 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. In fact, if 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 the participants collude to show a false view. However, with the assumption 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 to create disputes on the validity of views, allowing for 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, in the instances where 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 excluded. A case to apply pruning might be the case 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 bootstraping of light clients [19]. An interesting detail is that each view only comports the state and respective proofs on timeframe t_k. However, to guarantee that it is possible to validate the view, a pointer to the validity of the latest state before t_k should be available.

Are views suitable for representing on-chain data from different sources?

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 merge views belonging to different blockchains. The applicability is building a full picture of a participant’s activity on each network, but can also be leverage to disclose information according to an access control policy [50]. While selective access control to views has been explored, there is space to explore decentralized identity access control mecanisms to provide fine-grain access over views, leveraging the need to unify the different notions of identity that emerge from different blockchains. Zero-knowledge proofs can also be explored as a vehicle to prove facts on a ledger by disclosing limited information about such fact [57, 9]. We leave those interesting research paths for future work.

Another challenge refers to the problem of updating views. Let us assume one view generator creates a view at a certain time step t. The generated view will be different from its predecessor upon a new transaction. Generating a view can be a time-consuming, storage-intensive process. Thus, generating a new view accounting for only one transaction would be resource-consuming. To account for temporal differences across the same view, we can create a new view and then use a cryptographic accumulator to represent the state of a view (a sort of second-order view). This accumulator value (present of the proof field of a view) could represent incremental changes to the view and be managed according to a specific view management protocol.

5 Related Work

View-based data integration techniques have been summarized by Katsis al. [34]. *Global as View*, *Local as View*, and *Global and Local as View* are introduced as mechanisms to create an integrated view from local views. Global as View constructs a global view from each local view. Local as view constructs local views from a global view. Finally, Global and Local as View is a hybrid approach that generalizes previous techniques. Our approach follows a Global and Local as View 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 [8]. Next, we present how views can be used to enable blockchain interoperability.

The view as a standard data model for blockchain interoperability

Blockchain views are tools to enable blockchain interoperability because they promote data portability: a view is an intermediary standardized data format not dependent on a specific blockchain (agnostic) that can be translated across blockchains.

Abebe et al. [1] 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 in a decentralized way while allowing one to prove facts about a private blockchain. Our concept of view draws inspiration from this concept and generalizes it, providing algorithms for creating, merging, and processing views.

In [50], 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 for managing blockchain views in the context of blockchains providing partial consistency. The authors further define the concept of access control view, a list of keys used to decrypt part of a private subset. Our work subsumes this work, as it generalizes it to a wider range of applications.

In [40], the authors propose a unified programming model that abstracts concepts on ledgers (for example, payments, accounts, and conditions). The authors propose a programming language that compiles declarations (written in a domain-specific language) to operations in specific blockchains. With this model, the programmer can obtain standardized information from different blockchains. However, one cannot directly manipulate views, including creating or merging them.

There are some proposals on the industry and academia that propose general data models for cross-chain interaction, namely the Rosetta API, Quant Overledger's gateways [13, 54], Blockdaemon's Ubiquity API [15], Polkadot's XCMP [56, 46], Cosmos's IBC [36]. 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 also have the previous limitation and can only support blockchains created with Substrate and Tendermint, respectively. On the other hand, BUNGEE aims to create views that are independent of the underlying blockchains.

View as an abstraction to data isolation

Blockchain channels allow limiting the information available to each participant of the network, i.e., channels allow participants accessing a subset of the global ledger. One can consider the following as channels: shards, Hyperledger Fabric channels, Polkadot parachains, and others. The notion of view builds on top of channels to add selective access control (via the projection function).

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 [58]. As a result, parallelisation is possible, improving throughput and cutting communication overhead between nodes [35, 59]. Those nodes are thus 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.

Some permissioned blockchains contain privacy-enhancing features, typically using a form of sharding [2], that, as a consequence, can be reasoned about using the view. Hyperledger Fabric's supports both channels and private data. Still, on Fabric, Androulaki et al. propose multi-shard private transactions as a cross-shard privacy-preserving mechanism [3]. Hyperleger Besu [28] supports private transactions via groups – groups of participants entitled to see a subset of the global state – a specific view. Similarly, Canton and Corda also partition the global state according to permissions, creating views. In Polkadot, forks can exist due to the probabilistic nature of the block creation engine, BABE [45]. Thus, several temporary views may exist. In [50], views were used to provide fine-grain dynamic access control over private data in Hyperledger Fabric.

View applications

Besides 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 over different blockchains [49, 32]. In particular, a view is created and then possibly 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 [39], or cross-chain portfolio tracking [16], and even cross-chain security, by representing and monitoring cross-chain state [42], 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 formally present the concept of blockchain view, a foundational concept to understand partial consistency in blockchains. Views represent different blockchain participants' points of view, allowing reasoning about their different incentives and goals. We propose BUNGEE as 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 then take the retrieved states and create extended states, the basis for merging blockchain views. Different views (possibly from different blockchains) can be merged into a consolidated view. Finally, we discuss different aspects of BUNGEE, including decentralization, security, privacy, and its 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 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] E. Abebe, Y. Hu, A. Irvin, D. Karunamoorthy, V. Pandit, V. Ramakrishna, and J. Yu. Verifiable Observation of Permissioned Ledgers. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2021.

[2] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019. Publisher: VLDB Endowment.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery, 2018. event-place: New York, NY, USA.

[4] H. Bandara, X. Xu, and I. Weber. Patterns for blockchain migration. *arXiv preprint arXiv:1906.00239*, 2019. Publisher: Jun.

[5] R. Belchior. PhD Thesis Proposal - Blockchain Interoperability. Technical report, Instituto Superior Técnico, Sept. 2021.

[6] R. Belchior, M. Correia, and T. Hardjono. DLT Gateway Crash Recovery Mechanism draft 02. Technical report, Internet Engineering Task Force, 2021.

[7] R. Belchior, M. Correia, and T. Hardjono. Gateway Crash Recovery Mechanism draft v1. Technical report, IETF, 2021.

[8] R. Belchior, S. Guerreiro, A. Vasconcelos, and M. Correia. A survey on business process view integration: past, present and future applications to blockchain. *Business Process Management Journal*, ahead-of-print(ahead-of-print), Jan. 2022.

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

[10] R. Belchior, L. Riley, T. Hardjono, A. Vasconcelos, and M. Correia. Do You Need a Distributed Ledger Technology Interoperability Solution? *Techrxiv https://www.techrxiv.org/articles/preprint/Do_You_Need_a_Distributed_Ledger_Technology_Interoperability* Feb. 2022. Publisher: TechRxiv.

[11] R. Belchior, A. Vasconcelos, M. Correia, and T. Hardjono. Enabling Cross-Jurisdiction Digital Asset Transfer. In *IEEE International Conference on Services Computing*. IEEE, 2021.

[12] R. Belchior, A. Vasconcelos, M. Correia, and T. Hardjono. HERMES: Fault-Tolerant Middleware for Blockchain Interoperability. *Future Generation Computer Systems*, Mar. 2021.

[13] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. *ACM Computing Surveys*, 54(8):1–41, May 2021.

[14] J. Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[15] Blockdaemon. Ubiquity.

[16] Blockdaemon. Introducing Blockdaemon's New Staking Dashboard, Sept. 2021.

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

[18] V. Buterin. Chain interoperability. *R3 Research Paper*, 9, 2016.

[19] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias. SoK: Blockchain Light Clients. Technical Report 1657, 2021.

[20] R. Dijkman. Diagnosing differences between business process models. In *International Conference on Business Process Management*, 2008.

[21] EY. EY announces general availability of EY Blockchain Analyzer: Reconciler, 2022.

[22] J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In J. Katz and H. Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 291–323, Cham, 2017. Springer International Publishing.

[23] M. Graf, R. Küsters, and D. Rausch. Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric. 2020.

[24] M. Graf, D. Rausch, V. Ronge, C. Egger, R. Küsters, and D. Schröder. A Security Framework for Distributed Ledgers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1043–1064. Association for Computing Machinery, 2021. event-place: New York, NY, USA.

[25] T. Hardjono, A. Lipton, and A. Pentland. Toward an Interoperability Architecture for Blockchain Autonomous Systems. *IEEE Transactions on Engineering Management*, 67(4):1298–1309, Nov. 2020. Conference Name: IEEE Transactions on Engineering Management.

[26] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse Attacks on {Bitcoin's} {Peer-to-Peer} Network. pages 129–144, 2015.

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

[28] Hyperledger. Hyperledger Besu Ethereum client - Hyperledger Besu, 2019.

[29] Hyperledger. Private data — hyperledger-fabricdocs master documentation, 2022.

[30] Hyperledger Foundation. Hyperledger Fabric Private Data, 2020.

[31] Inside Bitcoins. What is Terra LUNA - Explaining the LUNA Crash, May 2022.

[32] Y. Jo, J. Ma, and C. Park. Toward Trustworthy Blockchain-as-a-Service with Auditing. In *ICDCS*, 2020.

[33] JP Morgan. Quorum White Paper, 2017.

[34] Y. Katsis and Y. Papakonstantinou. View-based Data Integration. In *Encyclopedia of Database Systems*, pages 3332–3339. Springer US, 2009.

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

[36] J. Kwon and E. Buchman. Cosmos whitepaper. *A Netw. Distrib. Ledgers*, 2019.

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

[38] A. Y. Levy. Logic-Based Techniques in Data Integration. In *Logic-Based Artificial Intelligence*, pages 575–595. Springer US, 2000.

[39] A. Li, G. Tian, M. Miao, and J. Gong. Blockchain-based cross-user data shared auditing. *Connection Science*, pages 1–21, 2021. Publisher: Taylor & Francis.

[40] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu. Hyperservice: Interoperability and programmability across heterogeneous blockchains. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 549–566, 2019.

[41] A. Lohachab, S. Garg, B. Kang, M. B. Amin, J. Lee, S. Chen, and X. Xu. Towards Interconnected Blockchains: A Comprehensive Review of the Role of Interoperability among Disparate Blockchains. *ACM Comput. Surv.*, 54(7), July 2021. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[42] I. Mihaiu, R. Belchior, S. Scuri, and N. Nunes. A Framework to Evaluate Blockchain Interoperability Solutions. Technical report, TechRxiv, Dec. 2021.

[43] B. Monnot. Visualising the 7-block reorg on the Ethereum beacon chain, May 2022.

[44] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[45] Petrowski, Joe. Polkadot Consensus Part 3: BABE, Dec. 2019.

[46] Polkadot. Cross-Consensus Message Format (XCM) · Polkadot Wiki, 2021.

[47] R3 Foundation. R3's Corda Documentation.

[48] P. Robinson. Survey of crosschain communications protocols. *Computer Networks*, 200:108488, 2021. Publisher: Elsevier.

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

[50] P. Ruan. LedgerView: Access-Control Views on Hyperledger Fabric. *https://www.comp.nus.edu.sg/˜ooibc/BlockchainView.pdf*, page 14, 2022.

[51] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen. Blockchain technology and its relationships to sustainable supply chain management. *International Journal of Production Research*, 57(7):2117–2135, Apr. 2019.

[52] W. F. Silvano and R. Marcelino. Iota Tangle: A cryptocurrency to communicate Internet-of-Things data. *Future Generation Computer Systems*, 112:307–319, Nov. 2020.

[53] C. team. Introduction to Canton — Daml SDK 2.1.1 documentation, 2021.

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

[55] P. Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1):285–287, 1996. Publisher: ACM New York, NY, USA.

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

[57] X. Yang and W. Li. A zero-knowledge-proof-based digital identity management scheme in blockchain. *Computers & Security*, 99:102050, Dec. 2020.

[58] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu. Survey: Sharding in Blockchains. *IEEE Access*, 8:14155–14181, 2020.

[59] M. Zamani, M. Movahedi, and M. Raykova. Rapid-Chain: Scaling Blockchain via Full Sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 18. ACM, 2018. event-place: New York, NY, USA.

[60] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt. Sok: Communication across distributed ledgers. In *International Conference on Financial Cryptography and Data Security*, pages 3–36. Springer, 2021.