Titanh: a Data Store Framework for Web3

Distributed Systems and Cloud Computing A.Y. 2023-2024

Emanuele Valzano
Computer Engineering
University of Rome Tor Vergata
Roma, Roma
emanuele.valzano@students.uniroma2.eu

Chiara Iurato
Computer Engineering
University of Rome Tor Vergata
Vittoria, Ragusa
chiara.iurato@students.uniroma2.eu

Alessandro Cortese
Computer Engineering
University of Rome Tor Vergata
Roma, Roma
alessandro.cortese@students.uniroma2.eu

***Abstract*—Traditional Web2 data stores often suffer from centralization, lack of transparency, and weak data ownership controls, making them vulnerable to censorship, data forgery, and limited user control. This project proposes a data store framework designed to offer the familiar abstractions of traditional data storage systems while achieving the goals of openness, transparency, decentralization, and resistance to forgery. The system is also designed to embed data ownership at the protocol level, as well as ensuring secure and verifiable control over stored content and providing resistance to censorship. This model enables new use cases in decentralized environments. Additionally, the system is built to support large-scale deployments and ensure high availability.**

I. Introduction

The system leverages two main technologies: Blockchain and a Decentralized File System. To counter the possibility of information being forged by outside agents, the immutability and append-only model provided by the blockchain are crucial. Everything executed on-chain is validated through a strict consensus protocol, certifying data ownership and acting as a consistent metadata layer for both data content and the system architecture itself. Since content cannot be stored directly on-chain due to limited block size, a separate system is needed to handle high loads of data at a large scale. A distributed file system fits this role, but only if having in mind the above characteristics. The blockchain will store only a limited amount of data, serving as pointers to the off-chain content. The system is built on top of the following systems: Substrate and IPFS.

A. Substrate

Substrate is an open-source framework offering foundational tools for building blockchains. It provides core libraries for essential blockchain components. Substrate is designed to abstract away much of the complexities associated with building blockchains. This allows the project to focus on functional and architectural aspects rather than blockchain low level details.

A core component within a Substrate based Blockchain is its Runtime, which consists of a set of modules, each responsible for specific functionalities. New modules can be developed and integrated into the runtime to extend the blockchain's capabilities. The runtime contains the logic executed in response to user transactions and is governed by strict consensus mechanisms. These modules are known as *Pallets*. The Runtime is compiled into *WebAssembly (WASM)*, and executed within an isolated, *WASM*-based virtual machine. *WebAssembly* ensures portability across different machines, allowing the runtime to run seamlessly on various platforms, which is crucial in decentralized and heterogeneous environments. The chain employs two distinct consensus protocols that operate asynchronously with respect to each other. The first, named *AURA*, is a block production algorithm, which selects the next validator node responsible for producing a new block, following a simple round robin scheme. To ensure liveness, even during network partitions, the block production process does not halt, which can lead to chain forks when different subsets of nodes produce competing blocks.

The second consensus protocol, named *GRANDPA* (a Byzantine Finality Gadget), implements a finality mechanism that resolves forks by selecting a single valid chain. This ensures that once a block is part of the finalized chain, it is considered final and immutable, preventing any further modifications. As a result, the Titanh Protocol guarantees deterministic finality, ensuring that the finalized chain remains consistent and irreversible. These aspects impact the consistency of the data store. The system generally follows an eventual consistency model, as users can only perform reads on produced or finalized blocks, rather than during block execution, which may include uncommitted write operations not yet visible. However, reads from a finalized state provide a highly consistent view, since all operations within the block are sequentially ordered and visible to all users, ensuring a consistent data view at the time of finalization. However, reading from a latest, non-finalized block may result in an inconsistent view due to potential chain rollbacks.

B. IPFS

Titanh employs the InterPlanetary File System (IPFS) for content storage due to its openness, decentralized nature, and censorship resistance, making it an ideal choice. As a widely adopted standard for file storage in Web3, IPFS is a peer-to-peer protocol for storing and sharing data in a distributed

system. It uses content-addressing to uniquely identify files through a content identifier (CID) derived from the file's cryptographic hash, allowing data to be located and retrieved across the network. To ensure data persistence and availability, IPFS utilizes a mechanism called pinning, which keeps content accessible. However, applications must run their own IPFS peers and handle pinning to ensure persistence, which can be a significant infrastructure burden.

Furthermore, IPFS does not inherently replicate data across nodes, as content is pinned only by nodes that are interested in it. A solution to this issue is provided by pinning services, which replicate content across multiple IPFS nodes (by means of pinning the associated CID) to ensure availability. However, these services tend to be centralized and often lack transparency, which runs counter to the principles of decentralization in Web3.

To address these limitations, Titanh integrates IPFS at its core, introducing replication mechanisms and utilizing the blockchain as a metadata layer for IPFS nodes. This approach ensures a more transparent and reliable system compared to centralized pinning services.

II. Context & Terminology

A. Context

To support common data store operations and embed ownership functionalities, we have extended the chain's runtime by implementing three custom *Pallets*, providing the foundation upon which the framework's APIs are built:

- *AppRegistrar Pallet*: This module allows users to create app instances that utilize the system and share ownership of these apps. It also enables users to subscribe to apps, granting them the rights to manage their own content on behalf of the app.
- *Capsules Pallet*: This module manages content metadata, facilitating both metadata upload and retrieval. It also implements sharing mechanisms.
- *PinningCommittee Pallet*: This *Pallet* manages the underlying architecture, specifically the nodes responsible for pinning content. Further details on this module are provided in the architecture description.

Regarding a *Pallet* implementation (which is part of the runtime), it involves defining various methods that are logically tied to the module, similar to an object-oriented structure. These methods, known as *extrinsics*, can be invoked externally by users and correspond to user transactions. However, extrinsics do not return values but instead trigger state transitions on the chain, emitting events during these state transitions, allowing external entities to subscribe to these events to verify the outcome of the method execution.

A key aspect to consider when implementing a chain's runtime is that every on-chain execution must complete within the block time; otherwise, the chain may halt. Our runtime defines a block execution time of 3 seconds, which should support various deployments and execution workloads.

Since we use Substrate, developers have the flexibility to integrate new functionalities or modify core *Pallets*, including those that implement different consensus protocols. Our project does not focus on these chain-level details, as the modularity of Substrate allows anyone to easily plug in their modules. For instance, while the current system uses AURA as the block production algorithm, which operates with a permissioned set of validators, permissionless environments are also feasible by integrating different *Pallets* and make slight changes to the node instance.

B. Terminology

- *Capsule*: It represents the smallest unit that is uploaded to the blockchain that identifies the content stored on IPFS. Each capsule holds the following metadata:
  - *Status*: The capsule status indicates whether a capsule is live or in the process of being deleted. This status is essential because when a capsule is deleted, all references to it must also be removed. However, since execution is bounded by block time, there is a risk that this process may not complete within a single block. The status reflects the different phases of capsule deletion. A capsule is considered logically deleted once a user with the appropriate permissions initiates the deletion transaction. The remaining garbage collection phases, which involve removing references to the capsule, can be performed by anyone, enhancing decentralization. Additionally, a garbage collection service (see architecture) may assist in completing this process.
  - *CID*: Identifies the actual content on IPFS.
  - *Size*: Represents the size of the underlying content.
  - *Ending Retention Block*: This represents the block at which the capsule's underlining content can be removed. However, the current framework implementation does not yet handle this scenario, meaning capsules' content is held indefinitely until users with the necessary permissions manually delete capsules entries. The system design, however, anticipates the inclusion of this feature in future iterations.
  - *Owners*: Indicates the chain accounts that own the capsule and can perform any operation on it. By default, a capsule is owned by the uploader. However, users can also give ownership of the capsule to other participants or share it with them.
  - *Followers Status*: Indicates who can subscribe to the capsule, representing users interested in the underlying content. However, not all capsules are open for following; the owners must enable this feature. There are three possible states:
    - *None*: No one has permission to subscribe to the capsule (default).
    - *Basic*: Anyone can follow the capsule as a Basic Follower, these are regular users with whom an application or owner is willing to share capsules.

    - *Privileged*: Only special accounts can follow the capsule, and these must be explicitly added by the owner, user subscription is not allowed. These followers are known as Privileged Followers, and are ”special” follower accounts that applications can distinguish from other followers, enabling a more fine-grained access control system certified by the blockchain. Privileged followers must be deliberately added by owner accounts.
    - *All*: Both Basic and Privileged followers are supported.
  - *App Metadata*: A sequence of app-specific encoded bytes that apps can use to identify a capsule, alongside an application ID.

  Each capsule is identified on the blockchain by a unique cryptographic hash, derived from the *App Metadata* field. The hash is constructed from three key components:
  - A constant string that is used specifically for capsule-related id computations. This distinction ensures that capsules and containers, even when sharing the same app metadata, will generate unique IDs.
  - The app identifier, which represents the application managing the capsule. This allows different applications to have identical app metadata but result in different IDs.
  - The remaining app-specific metadata.

  These components are concatenated and then hashed using the Blake256 cryptographic hashing algorithm. Since the process is entirely deterministic, users can retrieve a capsule simply by specifying its app metadata. This process will be abstracted through simple and common data store APIs, which will be described in a later section.

- *Container*: Containers are a collection of capsules. So, applications can upload several logically related blobs of data. Each container brings the following metadata:
  - *Status*: The container status defines if ownership is needed to attach or detach capsules or if it’s public. If ownership is set, container owners can remove capsules, which remain independently retrievable, and users must own both the container and capsule to add. In public containers, users can add or remove capsules as long as they own the capsule.
  - *Size*: Represents the number of capsules within the container.
  - Owners: Chain accounts that own the container.
  - App Metadata: A sequence of app-specific encoded bytes that apps can use to identify a container and describe the contained capsules, alongside an app ID used to manage the container.

  Like capsules, containers are identified on the blockchain by a cryptographic hash. This hash is computed similarly to capsules, but with a constant string prefix specific to containers, along with the app ID and app metadata. Each capsule within a container is uniquely identified by its key (raw sequence of bytes). This key points to a capsule identifier, which is necessary to retrieve the associated capsule metadata.

III. System Architecture

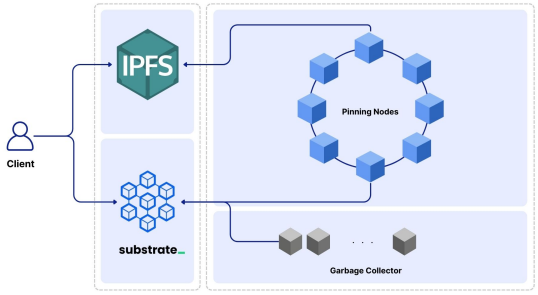


Fig. 1: System Architecture

The diagram illustrates the system architecture, showing the interaction between a client node, IPFS, and the Substrate chain’s gateways. It highlights how these interactions impact the system’s logical components: the Pinning Nodes and the Garbage Collector. The architecture follows an asynchronous model where clients first upload content to an IPFS gateway connected to the peer-to-peer IPFS network. Afterward, they submit a transaction to the chain (via a Substrate RPC node), uploading a capsule that contains all relevant metadata, particularly the CID returned by IPFS in the previous interaction. This process triggers the system components, Pinning Nodes and the Garbage Collector, which are subscribed to blockchain events emitted after a state transition (due to users transactions).

A. Pinning Node

At a high level, pinning nodes are the storage resources and implement replication on top of IPFS, enhancing fault tolerance and, in turn, increasing data availability and reliability. Content partitioning across nodes is handled using consistent hashing. Pinning nodes are organized in a ring structure, ordered by node IDs, with each node responsible for managing specific key ranges within the ring. Replication is achieved by pinning the CIDs that identify the content associated to a given key range across a designated number of successor nodes in a clockwise direction, based on the replication factor. Pinning nodes and keys share the same address space, where each key corresponds to a capsule identifier. Both capsule IDs and node IDs are computed using the Blake256 hashing algorithm.
Each pinning node communicate with its IPFS peers through an HTTP based API. These peers are responsible for hosting the actual content through IPFS. Additionally, each pinning node is linked to a blockchain validator node, binding it to a publicly known entity with a stake in the system. This association allows to identify the entity behind a pinning node, increasing the level of transparency. In light of this, the

blockchain serves as the metadata layer for the pinning nodes that form the ring. Instead of communicating directly off-chain, the nodes use the blockchain as middleware to monitor the ring's state. This eliminates the need for direct network communication between nodes, reducing communication overhead. Configuration parameters for the ring are stored on-chain and managed by the *PinningCommittee Pallet*, which handles the logic for managing the pinning ring.
The *Pallet* defines a genesis configuration, ensuring that the genesis block contains the necessary pinning ring parameters. The genesis configuration includes the following parameters:

- *Replication Factor*: Defines the number of pinning nodes that must pin an IPFS CID, ensuring the replication of the underlying content.
- *IPFS Peers*: Specifies the number of IPFS nodes associated with each pinning node, which are responsible for hosting the actual physical content.
- *Pinning Nodes*: The total number of pinning nodes per each validator.

The pallet defines methods to modify these parameters; however, the current implementation does not fully support changes to these parameters at runtime, as doing so would require a complete reconfiguration of the ring on-chain, including resetting and reconfiguring pinning nodes and restarting them from scratch.
Instead, join and leave operations can occur at runtime and are fully supported, allowing for incremental scalability. To support this, we have defined several extrinsics in the *PinningCommittee Pallet* to manage pinning nodes in the ring and integrated various mechanisms into the pinning node's internal architecture.

- *Node registration*: When a validator wants to register a pinning node, they send a registration transaction on-chain that includes all the IPFS peers composing the node. The registration is finalized when the validator signs and submits the transaction for the last IPFS peer. To prove ownership of each IPFS node, the transaction includes a signature of the validator's public key signed with the IPFS peer's private key. This cryptographically binds the IPFS peer to the validator and prevents replay attacks by other validators if the IPFS peer is later deregistered.
  Once registered, the pinning node is added to the ring at its correct position, determined by hashing the public keys of its IPFS peers to compute its ID. This registration triggers a chain state transition and emits an event to which other active pinning nodes are subscribed, allowing them to respond accordingly.
- *Node Leave*: The validator initiates a leave transaction specifying the pinning node to be removed and provides pointers to the key ranges it previously managed. Instead of including the actual key ranges—which would consume excessive block space—the transaction includes CIDs pointing to these key ranges stored on IPFS. Detailed information about this process is provided in the description of the Pinning Node's KeyTable.
  Upon completion of the leave operation, the runtime emits an event containing the departed node's ID and the pointers to the partitions. This allows other pinning nodes in the ring to identify the left node and retrieve the relevant data partitions representing the key ranges and their associated CIDs.

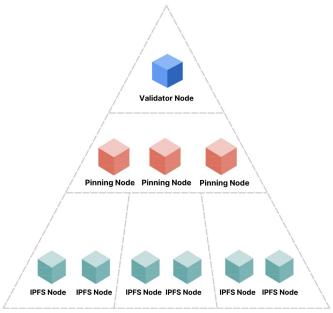


Fig. 2: Hierarchical Logical View

A graphical representation of the relationship between validator nodes, pinning nodes, and IPFS nodes is shown in figure 2. It is important to note that this is not a hierarchical architecture, but rather a logical view of the involved entities. Validator nodes never communicate directly with their pinning nodes, whereas pinning nodes communicate only with their IPFS peers. Communication between pinning nodes does not happen directly, as previously stated, but instead occurs indirectly through chain registration and leave operations.
Pinning nodes respond not only to join and leave operations but also to capsule uploads and deletions. They are subscribed to finalized events from the *Capsules Pallet*, specifically for capsule upload and deletion events. If a capsule identifier falls within the key range managed by a particular pinning node, it will pin the CID extracted from the upload event. In the case of a deletion event, the node will unpin the corresponding CID. It is important to note that all nodes are aware of one another, as the ring is stored on-chain, allowing each node to know its position within the ring, making key operations easy to perform.
It's important to note that since IPFS uses content-based addressing, different files uploaded by clients with identical content will result in the same CID, as IPFS considers them the same object. This creates multiple references to the same logically unique object identified by the CID. Therefore, when processing a capsule deletion event, the pinning node must check how many capsules point to the same CID. It's possible for multiple users to save the same content but with different application-specific metadata. The pinning node will only unpin the content if there are no remaining capsules in its key range associated with that CID.

B. IPFS HTTP API

To enable communication between system components and IPFS nodes, the system utilizes the IPFS HTTP API. This API allows client and pinning nodes to make HTTP requests to invoke specific methods on the local IPFS node, such as *add*, *pin add*, *pin rm*, and *get*, enabling content addition, pinning, unpinning, and content retrieval, respectively.

C. Substrate API

Components interact with the blockchain using a strongly-typed Substrate API. Each component needing to communicate with the blockchain does so through an RPC endpoint hosted by a full node connected to the peer-to-peer network. The API establishes a WebSocket connection to the node and communicates via the JSON-RPC protocol. This is used in client-side components, pinning nodes, and garbage collectors. JSON-RPC over WebSocket supports queries such as fetching the latest finalized or produced block. The API generates strong types from the chain's metadata, ensuring safe querying of state, signing and submitting transactions, and reading block events. In particular the system leverages Rust's type system to enforce type-safety, adapting to the chain's runtime for a consistent and reliable interface.

D. Garbage Collector

This component implements a garbage collection service. A user who owns a capsule can send a destroy transaction to initiate the deletion process. As previously mentioned, it is not guaranteed that all references to the capsule will be removed within a single block, since execution must be confined by the block time. Once the owner initiates the deletion, the capsule transitions to a "destroying" state. At this point, the capsule is considered logically removed, and if there are no other references to the underlying CID, the pinning nodes will unpin the CID, leading to the deletion of the content. However, metadata references to the capsule may still exist on-chain.
Each node in the garbage collection pool can manage specific key ranges, but if no range is specified, they can process all capsules. Nodes subscribe to finalized capsule destruction events, and if the capsule falls within their range, they begin the destruction process. Each transaction handled by these nodes triggers a new event, indicating whether the garbage collection process is complete or if on-chain garbage remains. This component is separate from the pinning nodes to increase decentralization and decouple system components. Even if many chain nodes store multiple versions of the state across different blocks, which causes deleted data to remain part of past blocks, it is important to remove unwanted data so that nodes can free up the underlying database within the latest blocks.

IV. Internal Architecture of a Pinning Node

We discussed what pinning nodes are and their roles, but now let's delve deeper into their specifics and examine their architecture.

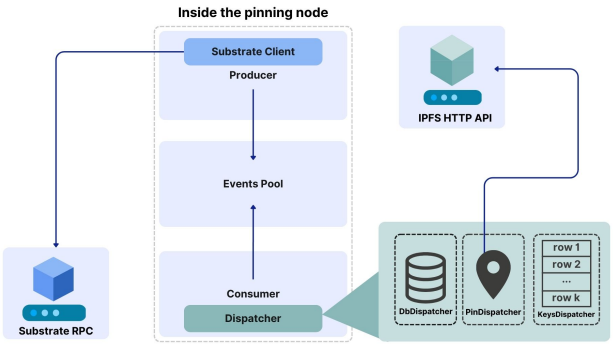


Fig. 3: Internal Structure of a Pinning Node

A. Pinning Node Components

A pinning node is structured into three main components: a *Producer* task, a *Consumer* task, and the shared *Events Pool*.

- *Producer*: This thread listens to finalized blocks on the blockchain to capture events related to the framework's custom *Pallets*. To achieve this, it embeds a Substrate Client, which runs an RPC stub that communicates with the Blockchain. The producer retrieves events from the *Capsules Pallet*, such as capsule upload, update, and deletion events. Additionally, it collects events from the *PinningCommittee Pallet* to monitor join and leave operations for nodes within the ring. These events are placed into an *Events Pool*, represented by a channel for further processing. The actual filtering to determine whether an event is relevant to the pinning node, based on its position in the ring, takes place on the consumer side. The consumer thread reads from the channel, and it dispatches it for further processing.
- *Events Pool*: This abstracts a channel, representing a pool of events that need to be processed and eventually dispatched in a first-in, first-out manner. The events in this pool are not the raw events emitted by the blockchain but have been tailored to fit the internal requirements of the pinning node. The types of events that constitute the pool are as follows:
  - *KeyedPinningEvent*: This event refers to an on-chain capsule event. It contains the capsule ID to which the event pertains and the corresponding pinning operation. The pinning operation indicates whether the action is a *pin addition*, *pin removal*, or *pin update*. In the case of a pinning update, the event includes both the old CID, which needs to be unpinned, and the new CID, which must be pinned.
  - *JoinEvent*: This event represents the entry of a new pinning node into the ring, as a result of the registra-

tion of the node. The event simply wraps the Node ID of the newly registered pinning node.
  - *LeaveNodeEvent*: This event signifies the departure of a pinning node from the ring. It contains the Node ID of the leaving node and a list of CIDs that identify the partitions representing the key ranges previously managed by the node.
  - *BlockBarrierEvent*: This control event is essential for creating a barrier between events from different blocks, enabling the consumer node to process events in batches, where each dispatch iteration corresponds to the processing of events from a single block. Additionally, it allows the event dispatcher to track the last processed block, providing a mechanism for checkpointing. Upon receiving this event, the node knows it has processed all events up to and including that block, allowing it to flush its state to disk and ensure the information is durable. In the event of a failure or node restart, the node can read the checkpoint and resume processing from the last processed block.

- *Consumer*: This component is responsible for consuming events from the *EventsPool*, continuously monitoring the underlying channel. The pool structure is shared between the producer and consumer tasks. As new events arrive, the consumer accumulates them into a batch. Once the block barrier event is received, the batch is dispatched through its embedded dispatcher, and the batch memory is cleared to make room for the next processing iteration. The dispatcher associated with the consumer handles the actual processing of the batch, transitioning the node state based on the received events.
  The dispatcher itself is composed of various fine-grained components, each specialized in handling specific operations. All dispatchers adhere to a common interface, ensuring consistency in how they process the events.
  - *DbDispatcher*: This component is responsible for dispatching block barrier events. It embeds a Database API to perform checkpointing operations, leveraging an embedded key-value database to flush the relevant node state.
  - *PinDispatcher*: A crucial component, responsible for executing pinning operations. It embeds an IPFS interface for HTTP-based communication with the node's associated IPFS peers. Specifically, it wraps multiple IPFS clients to enable load balancing across several node instances. Each client communicates with its corresponding IPFS node. The use of multiple IPFS nodes enhances fault tolerance by ensuring that if one node is offline, others can still handle the pinning operations, thereby increasing availability at the pinning node level. The load-balancing technique uses a uniformly distributed random number generator to select the client for adding new pins. If a selected node does not respond, the dispatcher attempts to contact another random instance, retrying up to a maximum number of attempts defined at the pinning node's startup. The underlying assumption is that at least one IPFS node will be available and selected by the RNG.
  - *KeysDispatcher*: Another important node component, in fact it handles operations within the key range handled by the node, depending on the pinning ring. In fact, it embeds two important objects: the Pinning Ring and the node's KeyTable.

    * *PinningRing:* This represents the pinning node's view of the ring at a specific block height, i.e., block number. The ring resides in the node's memory, and provides a global view of the ring's state, allowing the node to recognize all participants. Since the pinning node operates as a state machine, with its state evolving based on the events processed at a given block, the ring's state in memory reflects this progression. Therefore, the ring is consistent with the last processed block. This consistency is strong because the producer listens to finalized blocks, ensuring that everyone observes the same events in the same order, resulting in a unified and irreversible state across all nodes. However, the internal view of the ring mirrors the blockchain's state with slight latency. The ring is updated whenever the consumer processes a join or leave event, dispatching it to the *KeysDispatcher*.
    * *KeyTable*: This data structure abstracts the management of the node's key ranges. It is structured as a table of rows and columns. The number of rows corresponds to the replication factor, while the number of columns is unbounded and depends on the number of keys within each partition (row). Each row is implemented as a BTreeMap, where the map's keys correspond to capsule identifiers, and the values correspond to the associated CIDs. This structure allows efficient lookup of the CID for a given capsule key, which is essential for pinning operations. A BTreeMap is well-suited for this context because the ordering of keys simplifies tasks such as partitioning a row based on a barrier key during node join events or merging rows during node leave events. Rows in the *KeyTable* can be accessed by their row index.
      Since this structure resides in memory, it must be periodically flushed to disk, along with the block number it corresponds to, using the embedded key-value database. Each row of the *KeyTable* is serialized and stored under a separate database key in the format *"partition<idx>"*, where *idx* refers to the row index within the table. This method offers a compromise between storing each column in a separate key, which would compli-

cate reconstruction, and storing the entire table in a single key, which would be too resource-intensive. Moreover, since each entry in a row consists of 76 bytes (comprising a capsule key and a CID), a row can be easily stored under a single key in the database, especially if keys are well-distributed across the various ranges in the ring. Now that we've introduced the KeyTable, it's important to note that when a node leaves the ring network, it uploads the entire KeyTable to IPFS, uploading each row separately. This process generates different CIDs, which are included in the leave transaction, allowing other nodes to fetch the relevant data.

B. Pinning Node Execution Flow

The flow of a pinning node begins with the node bootstrap, which initializes the necessary components, followed by the execution phase.

Bootstrap

A high-level description of the steps involved during the bootstrap is outlined below:

1) *Reading the node checkpoint*: The checkpoint differs depending on whether the node is a newly registered node starting from scratch or an existing node restarting due to failure or maintenance.
   - If the node starts from scratch, the *KeyTable* is empty, and the ring information is retrieved by making an on-chain call at the latest finalized block. This ensures that the node is up to date with the latest consistent ring, which is then loaded into memory.
   - In contrast, if the node is restarting from an existing state (i.e., there is a checkpoint), it reads the ring state from the blockchain at the specific block height indicated by the checkpoint. This approach allows the node to avoid storing the ring on disk, as it can be retrieved on-chain based on the checkpoint's block number. The checkpoint also includes the *KeyTable*, which is up to date at the specified block number.
2) *Building the Producer and Consumer*: After checkpoint handling, both the Producer and Consumer are initialized, sharing the Events Pool. The Producer is constructed with the block number from which it must recover missed events. Whether the node is new or resuming from an existing checkpoint, its state is likely behind the blockchain, making the Producer responsible for recovering all missed events. Additionally, it must handle real-time finalized blocks. Therefore, the events to recover typically span from the recovery block up to, but excluding, the most recent finalized block, which is processed as a new block rather than part of the recovery. The Consumer is built with the *Event Dispatcher*, which requires the IPFS client, acting as the PinDispatcher, as well as the dispatcher responsible for database checkpointing operations. To construct the KeysDispatcher, we provide the ring, the KeyTable, and the corresponding block number.

Execution

In this phase, the node spawns the Producer task and consumes events via the main node thread. The Producer is designed to run indefinitely, continuously subscribing to finalized blocks, while the Consumer processes events retrieved from the shared pool.

Event production undergoes two distinct operations. First, the Producer starts by subscribing to finalized blocks to determine the upper bound for event recovery. Once the finalized block is "locked," the Producer begins fetching old events. There are two possible scenarios:

- *New Node (No Checkpoint)*: If the node is new and does not have a checkpoint, instead of fetching events block by block and producing everything—which could lead to inefficient operations, such as redundant pin and unpin actions for the same content due to capsule updates or deletions at later stages—it leverages its fresh state. The node downloads all currently live capsules and generates the corresponding *KeyedPinning* Events and produces them into the channel.
- *Restarting Node (Checkpoint)*: If the node is restarting from a checkpoint, rather than downloading all capsules, which would waste bandwidth and introduce overhead by comparing each downloaded capsule with those already in the KeyTable, the node simply recovers the events between the checkpoint block and the latest finalized block. The assumption here, for allowing the node to retrieve the events from blocks it has fallen behind on, is that the restarting node is not too far behind the current blockchain state. If the node has fallen significantly behind, it is recommended to reset its state entirely—deleting its content on IPFS and restarting from scratch. In this case, the node can download live capsules in real-time, which is more efficient and quicker than recovering a large backlog of events. The decision depends on how many blocks the node is behind.

Once the event recovery process is completed, the node begins producing events into the channel for the subscribed blocks, which are processed in real-time.

An important note to the above mechanisms is that since the Blockchain has a clock of some seconds, the pinning node will always be able to recoup the current finalized block, since its off-chain clock is much faster, even if it requires some network communication, this a simple RPC query, so latencies are not comparable to the blockchain block time. In addition, the download of capsules for a new node also is a process that does not constitute a bottleneck for the node to recoup the latest finalized block.

An important note regarding the above mechanisms is that,

since the blockchain operates on a block time interval of a few seconds, the pinning node will always be able to catch up with the current finalized block. This is because its off-chain clock is much faster, even when network communication is required. The latencies involved in these operations are negligible compared to the blockchain's block time. Additionally, the process of downloading capsules for a new node should not create a bottleneck, allowing the node to catch up with the latest finalized block.

The event consumer's execution corresponds to event dispatching, which involves fetching incoming events from the channel. The dispatching process is done in batches. Once the dispatcher receives an incoming batch, it iterates through the contained events and, based on the event type, performs one of the following actions:

- *KeyedPinningEvent*: If the event is a *KeyedPinningEvent*, it is dispatched to the KeysDispatcher. This component checks whether the key falls within its assigned key ranges by determining the partition index to which the key belongs, essentially returning the index of the KeyTable row where the key must be inserted. The algorithm for this process is outlined in the listings 1.
  If the index exists, the KeysDispatcher updates the KeyTable by either inserting or deleting the CID in the specified row, using the capsule key as the column index. This operation is determined based on whether the event is a pin, unpin, or pin update.
  Finally, the PinDispatcher is employed to either pin or unpin the CID, depending on the input event.
- *Node Registration Event*: Once a new node joins the ring, the node must first update its view of the ring and verify whether it should drop some keys, depending on the new node's position relative to itself. After potentially removing keys from the KeyTable by interacting with the KeysDispatcher, the related CIDs associated with those keys will be unpinned if there are no remaining references to them. This process is also presented as pseudo-code in the listings 2.
- *Node Leave Event*: When a node leave event occurs, the first step is to determine if the pinning node is affected by the ring update. If it is impacted, the node may need to add new keys to its KeyTable by retrieving the relevant row from the leaving node's KeyTable hosted on IPFS. The assumption is that the node leaving the network keeps its IPFS node running long enough for other nodes to fetch the necessary KeyTable rows. However, there are no guarantees that the IPFS node will remain online indefinitely.
  If a pinning node fails to retrieve its row—either due to its own prolonged outage and finding the other IPFS node offline, or due to the failure of the IPFS node hosting the KeyTable—the only option is to restart the node from scratch. In this scenario, starting from a checkpoint would cause the node to replay chain events and, when encountering the leave event, it would once again attempt to fetch a CID from IPFS, leading to an error if the content is no longer available.
  Once a row is successfully fetched, all underlying CIDs will be pinned on IPFS, and the fetched row will be used to update the node's KeyTable.
  A key consideration is that the KeyTable on the leaving node's IPFS is only up to date as of the block height of the corresponding keytable, which probably does not match the current finalized block. To address this, the pinning node must replay the block events that occurred between the block height of the leaving node's KeyTable and the current finalized block to ensure consistency.
  A more detailed view of these mechanisms is presented in the listings 3.
- *BlockBarrierEvent*: This event marks the end of the batch processing, resulting in the flush of the pinning node's state, useful in case of restarts or node failures. Specifically, the block number associated with the BlockBarrierEvent is saved to the database, indicating that events have been processed up to that point. Additionally, the KeyTable is flushed, as previously described. To improve efficiency, only the rows of the KeyTable that were modified during batch processing are flushed. This is achieved by tracking insertions and deletions to identify the affected row indices. Once these rows are flushed, the tracking flags are reset, allowing the next batch iteration to proceed.
  Moreover, during pinning operations, the system also tracks the reference counts for each CID to prevent incorrect removals. These reference counts are also stored in the database, where the CID is used as the key and the reference count as the value.
  The committed checkpoint is performed atomically by leveraging the Batch functionality offered by the database.

V. System's Metrics

This section focuses on the system-architecture-level metrics, while other measurements, such as the latency of the data store write and get operations, are discussed empirically in the Results section.

The system is primarily centered around two key metrics that impact not only the overall performance but also the reliability, particularly in relation to the time it takes for pinning nodes to pin the CIDs of the underlying content. The faster these nodes pin the content on their IPFS nodes, the quicker the content becomes fault-tolerant and assured to be available. In fact, there is a delay from the moment a user uploads content to an IPFS gateway (which doesn't pin the content indefinitely) to the time pinning nodes pin that content. This delay is primarily influenced by the block finalization time, as pinning nodes pin only CIDs associated to finalized blocks. However, several elements influence this delay:

1) The propagation time of the upload capsule transaction, which occurs immediately after the IPFS upload.

2) The finalization time, which is the predominant factor.
3) The latency involved in processing and dispatching capsule-related events within the pinning node.

A. Key Metrics

Given the above considerations, the two primary metrics of the system are:

1) *Throughput (capsules/second):* This refers to the maximum rate at which capsules can be processed by the system. The bottleneck for such processing is primarily determined by the blockchain network. The IPFS gateway, acting as a proxy for users to upload content, is not a bottleneck, as different concurrent users can utilize different gateways running on separate nodes.
   Given that, the three elements that impact the throughput are:
   - The total block weight
   - The upload extrinsic block weight
   - The block time

   The block weight refers to a metric in Substrate that measures the computational and storage cost of a transaction within a block. Substrate provides a benchmarking tool that allows developers to define test cases for extrinsics and execute benchmarks to determine transaction weights. Based on these benchmarks performed on the capsule upload method, it was found that such transaction consumes 0.02% of the total block weight, as shown in Figure 4. Therefore, approximately 5000 capsule upload transactions can fit into a single block. Given that a block is produced every 3 seconds, the theoretical upper bound for throughput is *1666 capsules/second*. However, this calculation assumes the block is filled solely with capsule upload transactions, which may not reflect real-world conditions but still provides a useful metric.
2) *Pinning End to End Latency:* This metric serves as both a tool to measure the pinning node's performance in processing and to compute the time related to the third point in the above list. It measures the end-to-end latency of a new batch entering the pinning node system. Since pinning nodes dispatch operations in batches, these operations are considered durable once a pinning node commits its checkpoint. Therefore, we measure the total time from the moment a batch enters the system to when it is dispatched and checkpointed.
   To track this metric, we introduced a *LatencyTracker* Event within the events produced by the pinning node's producer task. Since this event adds overhead to the system, it is only activated if specified in the node's execution parameters, and we do not recommend its use in production, but rather for analysis purposes.
   When the *LatencyTracker* is active, the pinning node includes the event at the head of each batch, containing the current system time (representing the batch's entrance time). As the consumer receives the *LatencyTracker* Event, it records the wrapped time and once the *BlockBarrier* Event is consumed and the checkpoint is committed, the system calculates the elapsed time. This gives the total end-to-end latency for the batch, which is logged for analysis.

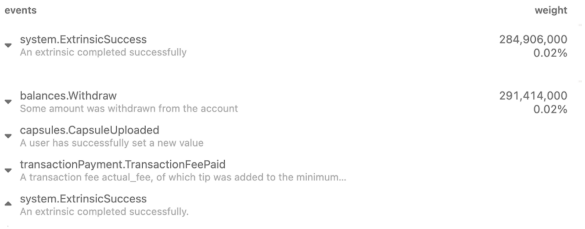


Fig. 4: Upload Extrinsic Weight

VI. Framework APIs

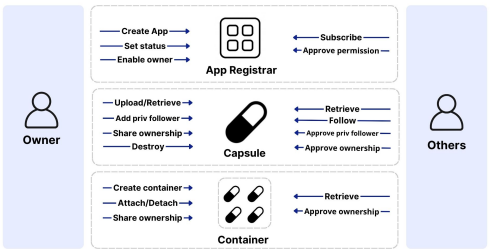
To interact with the APIs provided by the framework, a user must possess an on-chain account, which requires holding a private key. Additionally, the on-chain account executing these API calls must have sufficient balance to cover transaction fees. However, the framework deployment may opt to remove the *Pallet Balances*, which handles the token-related logic and transaction fees. This choice depends on the specific needs of the framework deployer and will impact the applications built on top of it.

A. Titanh Core

At its core, the framework supports several functionalities, which are implemented by the following Pallets:

- *AppRegistrar Pallet*
- *Capsules Pallet*

The APIs exposed by these *Pallets* can be accessed directly, providing fine-grained control over the data store system. A full description of the methods, illustrated in the image below, can be found in the Appendix Section.



B. Levels of Consistency

In the context of interacting with the Blockchain, control can be returned to the caller at different stages, as it is closely tied to the transaction lifecycle. We have:

- *wait async*: offers the least stringent level of confirmation. When invoked, it proceeds without waiting for any confirmation from the blockchain, apart from transaction validation and inclusion in the transaction pool. This

approach can be beneficial in scenarios where speed is a priority.

- *wait block inclusion*: provides a moderate level of confirmation. It waits for the transaction to be included in a block. This level is suitable for applications that require some degree of assurance that the transaction has been processed, although it does not guarantee that the transaction is permanent or cannot be reverted.
- *wait block finalization*: represents the highest level of confirmation. Waits for the transaction to be finalized, providing the strongest assurance that the operation is immutable and permanently recorded on the blockchain. This level is essential for applications that require strict data integrity and reliability, as it ensures that the transaction is beyond dispute.

It's important to consider how these choices impact the datastore in terms of both read and write operations. This choices mirrors common distributed consistency models, but differs slightly from the standard definition, due to presence of blocks:

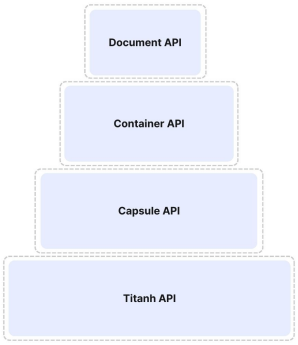
1) wait async resemble **eventual consistency** models, where changes will eventually propagate. In Titanh, this means that when a put operation is performed, if someone reads from the same block as the put, before the block production, the capsule won't be visible. However, once a read is made after the block is being produced, everyone can see the most recent update to the capsule, ensuring a stronger consistency. It's important to note that if a network partition occurs, there's a possibility of a rollback, due to chain forks, meaning the block might never be appended to the chain, and as a result, the content may not be pinned.
2) Block inclusion is similar to **weak consistency**, offering some level of confirmation but without finality. Again, if a network partition occurs, the content might not be pinned.
3) Finalized blocks provide **strong consistency**, guaranteeing that once a transaction is finalized, any read at that block will result in the same value.

All these considerations assume that an IPFS gateway is always reachable when pinning nodes must pin the content. If the pinning nodes are unable to contact the node behind the gateway, the client will need to refresh the data. In such cases, the blockchain won't return an error, and the transaction will still be recorded on-chain, potentially leading to inconsistent content. To mitigate this issue, a replicated gateway setup can be deployed, significantly improving the system's reliability.

The most common implemented datastore APIs support these different consistency levels, allowing for flexibility depending on the application's needs.

C. Higher Level APIs

In addition to the Core Titanh APIs, we provide clients with higher-level APIs that follow common data store patterns, offering familiar operations such as *put* and *get*, with support for various consistency models. The sequence diagrams below illustrate the underlying execution flow and interactions between system components for the *put* and *get* operations. A detailed description of these APIs, along with each method, can be found in the Appendix Section.



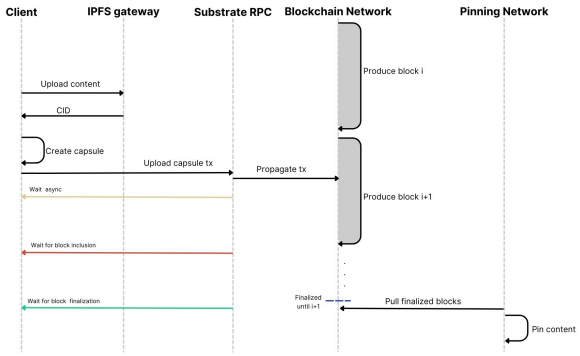


Fig. 5: Put Sequence Diagram

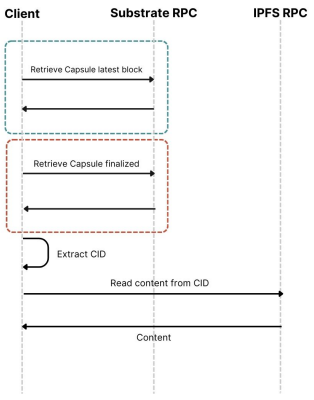


Fig. 6: Get Sequence Diagram

VII. Results And Empirical Evidence

A. Deployment

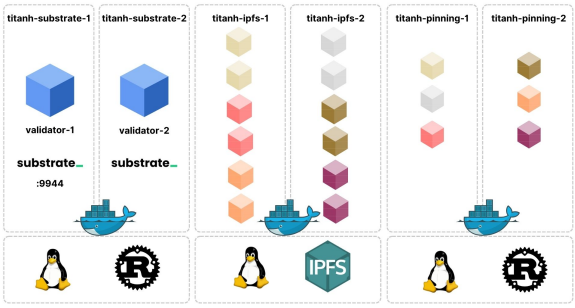


Fig. 7: Deployment of the System

For experimental purposes, we emulated the architecture using Docker and Docker Compose. As illustrated in the above image, the architecture was deployed across six containers. The more relevant deployment configuration parameters are the following:

- *REPLICATION_FACTOR*: 2
- *IPFS_PEERS*: 2
- *PINNING_NODES*: 3

The full deployment description is provided in the Appendix Section.

B. Latency Analysis Across Consistency Levels

In this section, we measured the latency of the most common datastore operations from the user's perspective. Specifically, we recorded the time before invoking the APIs and then measured the elapsed time after the API calls were completed. The APIs were invoked across all consistency levels to assess their impact on the execution time. Furthermore, we measured this time for different content sizes to analyze how increasing byte sizes impacts the overall performance.

We developed a test case program that randomly fills buffers of different sizes, based on the specified parameters. The program is designed to measure the execution time of **put**, **get**, and **put_batch** operations across various consistency levels.
We executed the operations within 10 iterations and calculated the average latency. The buffer sizes used range from 1 MB to 90 MB, with increments of 10 MB.
The test case was executed in a host process that communicates with the emulated architecture through the Docker-exposed ports.
Since the entire infrastructure is emulated on a single machine, the following results should be interpreted cautiously, as they do not account for network latencies between components. Additionally, the host system is under significant load due to running numerous processes, which may influence the results, particularly due to the operating system's scheduling of these components. However, these results provide an indicator of how the system performs under these conditions and allow us to measure its consistency in relation to the operations.

Stronger Consistency Levels

- For the ***put()*** operation, under both Weak and Strong consistency levels, we expect that most of the execution time will be spent waiting for a block to be produced or finalized, depending on the selected consistency level. Unless the content size is so large that the time required to write it exceeds the block latency.
  - For the weaker consistency level: Referred to as the *Committed Level* in the plots, we expect the time to be bounded by the block production time, falling between 0 and 3 seconds. The exact time depends on when the transaction arrives at the block producer with respect to the block production phase.
  - For the strongest consistency level: Referred to as *Finalized* in the plots, we expect the time to fall within a range based on the block finalization time, which typically takes longer.

  As shown in the charts in Figure 8 and 9, these expectations are fully met.

- For the ***get()*** operation, the results obtained from the two consistency levels are not particularly interesting to compare, as the execution times are nearly identical. This behavior is expected, as the difference between the two consistency levels lies not in the response time but in the type of state returned to the client. The choice of which operation to invoke depends on the freshness of the data the client requires, especially after write operations. If a client needs the result sooner, the weaker model (based on the latest block) will yield a faster response.

  The charts in Figure 10 and 11 confirm these expectations. Additionally, we can observe that execution times increase with the size of the content being fetched.

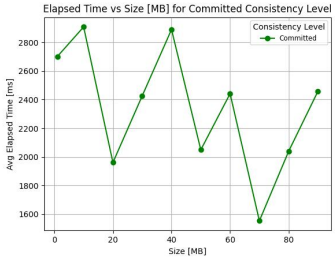


Fig. 8: **weak** `put()`

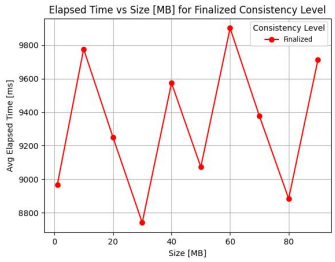


Fig. 9: **strong** `put()`

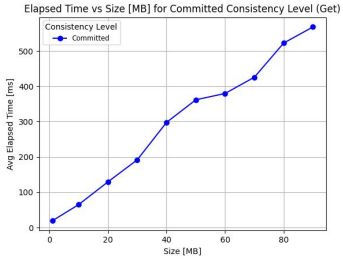


Fig. 10: **weak** `get()`

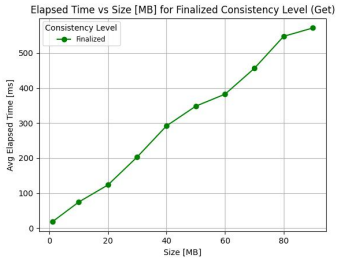


Fig. 11: **strong** `get()`

Eventual Consistency Level

This consistency level refers only to the write-based APIs. As we can see from the below chart, execution times significantly decrease compared to the other consistency models. This is of course because the transaction does not wait for confirmation; as soon as it is deemed valid and enters the transaction pool, the control is returned to the client.

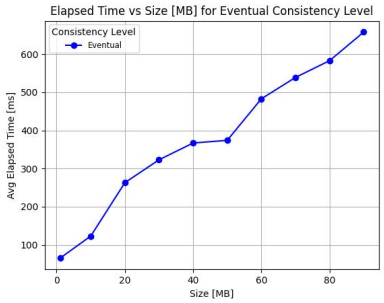


Fig. 12: **eventual** `put()`

Note that *put_batch* operations were performed across all content sizes. The table below presents the obtained results. The execution times are of course higher because this operation uploads all related content within the batch, totaling 451 MB. Notably, the committed consistency model is no longer constrained by the block time but instead exceeds it, primarily due to the additional time required to upload the larger content to IPFS before the on-chain transaction is processed.

Size	Consistency level	Avg elapsed time [ms]
472907776	Eventual	2503
472907776	Committed	3797
472907776	Finalized	12809

`put_batch()`

Considerations and Conclusions

As a result of these experiments, we can conclude that the datastore is well-suited for applications that do not require highly sensitive latency, but rather those that can tolerate medium-latency tasks. A crucial point to note is that the observed latencies are strongly tied to the consensus protocol employed by the blockchain, which represents the true bottleneck. These protocols significantly influence the block production and finality times, so it is highly recommended to integrate the most appropriate consensus mechanism, particularly one that is optimized for finality.

However, users can still leverage the eventual consistency model to develop high-speed applications, specifically for write-intensive workloads. For read operations, even under the weaker consistency models, there is still a dependency on block production, which introduces some delay.

In conclusion, the datastore is better suited for write-heavy applications when latency sensitivity is a concern. For read-heavy tasks, it is more appropriate for medium-latency applications where response time is less critical.

Future iterations of the framework, as detailed in the *Future Work Section*, may improve the latency concerns.

VIII. Use Cases

There are several potential use cases that can be implemented within the Titanh Framework. The framework includes an example application that demonstrates how to store university certificates, leveraging the framework's ownership features and its resistance to forgery. In this use case, certificates can be shared with hiring companies, offering a transparent source of truth.

For instance, a university can use an on-chain account to transfer ownership of the certificate to the student. This allows anyone, such as potential employers, to easily verify the digital validity of the certificate, reducing the need for trust in the candidate. Instead, trust is placed in the university as the issuer, which can be verified directly on-chain. This approach simplifies the hiring process by ensuring certificates are authentic and tamper-proof.

Other use cases include:

- Public verifiable storage: Storing content that can be easily verified by anyone to confirm its authenticity and source.
- Pinning Services: Acting as a decentralized pinning service on IPFS, ensuring availability to user files.
- Decentralized data ownership: Ensuring users retain full control over their data.
- Digital Identities: Storing and verifying users' digital identities with resistance to forgery.
- Backup system: Saving old data from other distributed systems, ensuring it is always maintained and never deleted. This would provide an immutable backup solution where data remains available for long-term storage and can always be verified for authenticity.

IX. Future Work

This work represents the first iteration of the framework, and as such, there are several areas for improvement:

- Virtual Nodes: Future iterations of the framework could introduce virtual nodes, allowing multiple pinning nodes to share the same IPFS resources. Currently, reusing an IPFS node for multiple pinning nodes is not permitted as it would break the replication factor mechanism.
- Failure Detection: Currently, the framework does not support failure detection for pinning nodes. If a node fails, other nodes continue assuming it is still active in the pinning ring. To address this, the framework could be extended by introducing a *PinningRingHeartbeat Pallet*. This would allow pinning nodes to send periodic heartbeat messages using the blockchain as middleware. If a node fails to send a heartbeat within a specified threshold, the chain's runtime could automatically remove it from the pinning ring, improving reliability and fault tolerance.
- Multiversioning: The framework currently supports versioning only at the metadata level, leveraging blockchain immutability. However, content itself is not versioned—once deleted, it is unpinned and irretrievable. Future versions could introduce multiversioning for content.
- Content retention: The framework has established the foundation for this feature, though it is not yet supported. It can be included in future iterations to enable automatic removal of unused or obsolete content.
- Storage Proofs: Pinning nodes could periodically send cryptographic proofs to demonstrate they are storing the content, allowing anyone to verify that they are performing their role as expected. This would reduce the need to trust pinning nodes blindly and enhance the transparency and reliability of the system.
- Game-Theoretic Mechanisms: Integrate game-theoretic mechanisms to incentives proper behavior and penalize misconduct within the network. This could include rewarding pinning nodes for maintaining content availability and imposing penalties for failing to store or deliver content as promised.
- Layer 2 System: Implement a Layer 2 system that functions as a caching mechanism within the network. This system can aggregate user transactions, providing a faster response by returning the resulting state, while asynchronously sending the individual transactions to the underlying blockchain. However, this approach introduces security considerations that must be carefully addressed.

APPENDIX

A. Pinning Node Algorithms

```
fn key_node_partition(ring, key, node_id) {
    next_node_idx = ring.binary_search_closest_node(key)
    ring_size = ring.len()
    sum = next_node_idx + ring.rep_factor()

    replicas = []
    if sum < ring.len() {
        replicas.extend(ring[next_node_idx..sum])
    } else {
        diff = sum - ring_size
        replicas.extend(ring[next_node_idx..ring_size])
        replicas.extend(ring[0..diff])
    }

    partition_idx = replicas.find(node_id)

    return partition_idx
}
```

Listing 1: Key Row Index Algorithm

```
fn join_node(node, ring) {
    idx = ring.insert(node)
    dist = ring.distance(idx, current_node_id());
    if dist <= ring.rep_factor() {
        row_idx = dist - 1;
        barrier = node;
        rm_row = partition_row(row_idx, barrier)
        try_to_unpin(rm_row)
    }
}
```

Listing 2: Node Join

```
fn leave_node(node, current_block, current_event_idx,
     ring) {

    dist = calculate_distance(current_node_id(), node);
    ring.remove(node);
    if(dist <= replication_factor()){
        let row_idx = dist - 1
        merge_rows(row_idx)

        cid_idx = ring.rep_factor() - dist
        cid = node.cid_of(cid_idx)

        start_block = node.key_table_height() + 1
        replayed_events = replay_blocks(start_block,
             current_block, current_event_idx)
        foreach event in replayed_events:
            update_key_table(row_idx, event)

        batch = build_batch(replayed_events)

        fetched_row = update_pinning(cid, batch)

        extend_keytable(fetched_row)
}
```

Listing 3: Node Leave

B. Pallet extrinsics

App Registrar

- `create app`: a user is able to register an application on the blockchain, receiving an AppID essential for the capsule upload.
- `set subscription status`: the status can be set to:
  - `Anyone`: open for public subscription
  - `SelectedByOwner`: restricted to users specifically chosen by the owner
- `enable owner`: grants shared ownership of an app to designated users.

Capsules

- `upload capsule`: It allows uploading a new capsule associated with an existing app, verifying that the user has the app related permissions. Furthermore, the uploader has the option to specify another owner, thus directly transferring the capsule's ownership, otherwise, the uploader is the owner.
- `approve capsule ownership`: approves ownership of a capsule, allowing to become one of the owners of the specified capsule.
- `share capsule ownership`: allows an owner to share ownership of the capsule with another user. The other user must approve the shared ownership.
- `set capsule follower status`: sets the status of followers for a capsule, changing the behavior related to who can follow it.
- `follow capsule`: allows a user to follow a capsule, as long as the capsule follower status is consistent with this operation.
- `update capsule`: allows to update a capsule.
- `add privileged follower`: allows to add a privileged follower, allowing the off-chain application to treat it as intended.
- `approve privileged follow`: it is an approval of the previous add operation.
- `start destroy capsule`: allows to start the deletion process of a capsule.
- `create container`: Allows the creation of a new empty container, and like the behavior for capsules, it optionally allows specifying another owner.
- `attach/detach capsule`: it attaches/detaches a capsule within a container, providing the container ID and an input key that will identify the capsule in the container.
- `share ownership`: operate under the same conceptual functionality as capsules.

C. Higher Level APIs

Capsules APIs Such APIs abstracts the *Capsules Pallet*, by implementing on top of it commonly used data Store APIs. All of the capsule-related APIs, when dealing with write operations, support all three levels of consistency.
To utilize the Capsules API developers must provide an IPFS HTTP URL (acting as the IPFS gateway) and specify the application ID for the session. Additionally, if the user wishes to perform methods involving write operations on the datastore, they must provide their account's private key. The API supports the following operations:

- `put(Id,Value)`: When a client call this method, it must specify an id and a value, accordingly to an interface for serialization. This process consists of two main steps: first, the data (value) is uploaded to IPFS, which returns a CID that is used to create the capsule along with its metadata. Next, the client sends the transaction to the blockchain.
- `get(Id)`: Conversely, when a client wishes to retrieve a previously stored content, they only need to specify the ID. The client can choose to read the content based on the latest produced block or the finalized one.
- `putbatch(CapsuleBatch<Id,Value>)`: This method enables the simultaneous upload of multiple (id, value) pairs grouped in a batch, following an all-or-nothing policy.
- `update(id, value)`: This method enables the updating of existing content. The new content is uploaded to IPFS, and the returned CID replaces the old one by including it in the *Capsules Pallet* update transaction.
- `remove(Id)`: this methods allows to remove the content identified by the Id.

There are additional APIs that serve to simplify the sharing process, specifically for owners and followers.

Container APIs

This API is primarily designed to support the higher-level *DocumentAPI*, which facilitates the management of datastore documents. Therefore, we recommend using the document-centric APIs for most operations. For those requiring more granular control, the Titanh Core and Capsules APIs are suggested.

Document APIs

This API represents the higher level abstraction of the datastore, enabling the creation and management of documents. Documents also support all three consistency levels for write-based operations. The available methods are outlined below:

- `createdocument(Id)`: it allows to create a new Document, based on the provided Id, that identifies the document. Under the hood, this operations creates a new container. This method does not support different consistency levels, primarily because users typically create documents once and then perform operations on the same object. As a result, we only offer the highest level of consistency.
- `insert(key, value)`: this method allows the addition of a new field to the document, with the field (value) identified by the input key. To perform this operation, we first upload the capsule, generating a custom ID by hashing the document ID along with the key that identifies the field. Afterward, the uploaded capsule is attached to the underlying container. These operations are executed atomically, adhering to an all-or-nothing policy.
- `read(key)`: This method retrieves the field identified by the key in the specified document. It first performs an on-chain query to fetch the capsule ID within the container based on the input key. The actual capsule is retrieved by specifying the obtained capsule ID. Finally, using the CID within the capsule, the content is fetched from IPFS. This read operation can be performed either on the latest produced block or the finalized one, depending on the application's consistency requirements.
- `remove(key)`: This method removes an existing field from the container based on the identifying key. The removal process first detaches the capsule from the container and then deletes the corresponding capsule entry. As with insertion, the operation is performed atomically.

D. System's Deployment Details

- The first two containers run the blockchain validators, with the first node acting as a bootnode for the second node. The first validator node owns the pinning nodes associated with the *titanh-pinning-1* container, and, consequently, all underlying IPFS peers in the *titanh-ipfs-1* container. The same applies for the other validator, but for the remaining containers.
- The following two containers are responsible for executing the IPFS peers associated with the pinning nodes. These nodes are configured with specific parameters to enable the correct IPFS daemon execution. In particular, we provided the peers' private keys, PeerIDs, and ports associated with the IPFS HTTP API used for communication with the pinning nodes.
- The remaining containers handle the pinning node execution. Key configuration parameters include:
  - pinning_instances = 3
  - validator_seed: used for signing the registration/leave transaction
  - chain_rpc
  - ipfs_http
  - replication_factor = 2
  - failure_retry = 2