Linera: a Blockchain Infrastructure for Highly Scalable Web3 Applications

Version 2 – August 16, 2023

Abstract

We present Linera, a blockchain infrastructure that aims to support the most demanding Web3 applications by providing them with predictable performance, security, and responsiveness at the Internet scale. To do so, Linera solves the blockspace scarcity problem by introducing a new, integrated, multi-chain paradigm built upon elastic validators. Linera puts users at the center of the protocol by allowing them to manage the production of blocks in their own chains—called *microchains*—for optimal performance. To help Web3 developers make the most of the Linera infrastructure, we have developed a rich, language-agnostic, multi-chain programming model. Linera applications communicate across chains using asynchronous messages. Within the same microchain, applications are composed using synchronous calls and ephemeral sessions (aka resources). The initial SDK of Linera will target Rust programmers, thanks to the Wasm virtual machine. The Linera infrastructure is based on delegated proof of stake. It will ensure robust decentralization using state-of-the-art economic incentives and auditing at scale by the community. [1]

[1] Legal Disclaimer: This document and its contents are not an offer to sell, or the solicitation of an offer to buy, any tokens. We are publishing this white paper solely to receive feedback and comments from the public. Nothing in this document should be read or interpreted as a guarantee or promise of how the Linera infrastructure or its tokens (if any) will develop, be utilized, or accrue value. Linera only outlines its current plans, which could change at its discretion, and the success of which will depend on many factors outside of its control. Such forward-looking statements necessarily involve known and unknown risks, which may cause actual performance and results in future periods to differ materially from what we have described or implied in this document. Linera undertakes no obligation to update its plans. There can be no assurance that any statements in the document will prove to be accurate, as actual results and future events could differ materially. Please do not place undue reliance on future statements.

Contents

1 Introduction **3**
1.1 The need for predictable performance and responsiveness in Web3 3
1.2 The blockspace scarcity problem 3
1.3 Shortcomings of existing approaches 3
1.4 Our mission 4
1.5 Overview of the project 4
1.5.1 An integrated multi-chain system with elastic validators 5
1.5.2 Making multi-chain programming mainstream 5
1.5.3 Robust decentralization for elastic validators 6

2 The Linera Multi-Chain Protocol **6**
2.1 Participants: users, validators, chain owners 6
2.2 Security model 7
2.3 Notations 8
2.4 Microchains 8
2.5 Cross-chain requests 10
2.6 Chain states 10
2.7 Block execution 11
2.8 Client/validator interactions 12
2.9 Extensions to the core protocol 13

3 Analysis of the Multi-Chain Protocol **15**
3.1 Responsiveness 15
3.2 Scalability 15
3.3 Security 16

4 Building Web3 Applications in Linera **18**
4.1 Creating applications 18
4.2 Multi-chain deployment 19
4.3 Cross-chain communication 19
4.4 Local composability 20
4.5 User authentication 20
4.6 Ephemeral chains 21

5 Decentralization **21**
5.1 Delegated proof of stake 21
5.2 Auditability 21

6 Conclusion **22**

A Cross-chain communication **28**
A.1 Messages and inboxes 28
A.2 Cross-chain requests and outboxes 29

1 Introduction

1.1 The need for predictable performance and responsiveness in Web3

Thanks to blockchain technologies, the next iteration of the Internet, Web3, will empower users with a new generation of asset-aware applications and give them more democratic control over the digital economy. However, developing Web3 applications with a great user experience is currently a challenging task. One of the issues is reliability and responsiveness at scale: when too many users are active, blockchains may stop responding or demand punishing fees. In general, application developers want their infrastructure programming interfaces to be easy to use and predictable, disregarding the traffic caused by other applications. Centralized API providers [30] have been proposed to facilitate programming on top of popular blockchains, but such providers need to be trusted and will not improve the performance and the fees of the underlying blockchains. Linera aims to close the gap between centralized and decentralized applications by delivering a blockchain infrastructure that guarantees performance and responsiveness at scale.

1.2 The blockspace scarcity problem

The main reason why traditional blockchains have unpredictable worst-case outcomes in terms of fees and delays can be explained as the *blockspace scarcity* problem. Namely, in a blockchain composed of a single chain of blocks, users must compete to have their transactions selected into the next block. Yet, at the same time, the production rate and the size of blocks are limited by the performance of the consensus protocol, the network, and the execution layer. As a result, during a peak of traffic (say, an NFT airdrop), users may be outpriced by others or be delayed for long periods of time—during which the infrastructure is effectively unavailable to them [21].

1.3 Shortcomings of existing approaches

Unsurprisingly, many blockchain infrastructures have been proposed over the years with scalability improvements in mind. We provide here a high-level summary of the most common approaches, without attempting to be exhaustive.

Faster single chain. The production rate of blocks in a single chain is typically limited by the data propagation delay between validators [18]. Historically, block size has been the first parameter to be adjusted to maximize transaction throughput in function of the security requirements and the network constraints [18, 20]. Thanks to recent advances in BFT consensus protocols (*e.g.* [22]), nowadays the new bottleneck for the transaction rate appears to be the sequential execution of transactions rather than consensus ordering.

Anticipating that many transactions contained in a block should be independent in practice, several recent projects have developed architectures able to execute a subset of transactions in parallel on several processing units [19]. While this certainly results in higher transaction rates, such systems are still characterized by a maximum number of transactions per second in the low 6 digits. Moreover, the effective transaction rate greatly depends on the proportion of transactions that are actually independent in each block [26]. Altogether, this makes it impossible to guarantee fees and/or delays in advance for a user without any assumption about the activity of the other users.

Lastly, in a high-throughput chain, auditing validators is made harder by the combination of CPU requirements for execution and networking requirements for data synchronization. Concretely, the sheer number of sequential transactions may prevent members of the community with only commodity hardware from replaying transactions fast enough to verify the work of validators in a meaningful way [24].

Blockchain sharding. Another popular direction to address blockchain scalability has consisted in dividing the execution state between a fixed number of parallel chains, each being run independently by a separate set of validators. This is called *blockchain sharding*.

While this approach is still being continuously improved, it has historically suffered from several challenges. First, using separate sets of validators creates a security tradeoff in so far as an attacker may selectively attack the weakest set in the system (*e.g.* to mint coins). Second, reorganizing the *shards*, *i.e.* the way user accounts are distributed across chains, is a complex operation that necessitates extensive network communication [33]. Lastly, when the number of shards is increased to support additional traffic, so does the amount of cross-chain messages that need to be exchanged [26]. In a system where each shard has a separate set of validators, cross-chain messages create significant delays that ultimately cancel out the effect of adding new chains [31, 33].

Rollups. Finally, a popular approach to solve blockspace scarcity has been *rollup* protocols, either *optimistic* or based on *validity proofs* (aka *ZK rollups*) [11]. At a high-level, both optimistic and validity ("ZK") rollups consist of a layer-2 protocol that builds a sequence of large blocks, meant to be executed, compressed and confirmed on layer 1. Unfortunately, the process of confirming transactions on layer 1 takes a long time in both cases. Optimistic rollups must wait several days to allow for dispute resolution. Validity rollups must compress many layer-2 transactions at a time to pay for the layer-1 gas. In practice, gathering enough layer-2 transactions, computing a validity proof, and archiving transactions to enforce rigorous data availability takes several hours per layer-2 block.

Long layer-1 confirmation times may encourage certain users to accept a security tradeoff and trust the finality of layer 2 for certain applications. In general, rollups must be trusted to carry on the protocol (*i.e.* for *liveness*) and to select transactions fairly (see Miner Extractable Value [15]). This concern is visible in the recent efforts to design decentralized rollup protocols [29].

1.4 Our mission

Motivated by these observations, the Linera project was created to develop a new type of Web3 infrastructure based on three key principles:

(i) Build a secure infrastructure with predictable performance and responsiveness — by operating many chains in a single set of elastic validators;

(ii) Enable a rich ecosystem of scalable Web3 applications — by working on a new execution layer to make multi-chain programming mainstream;

(iii) Maximize decentralization — by ensuring that elastic validators are optimally incentivized and audited at scale by the community.

1.5 Overview of the project

Linera is dedicated to delivering the following innovations to the blockchain community.

1.5.1 An integrated multi-chain system with elastic validators

To fulfill our vision of a Web3 infrastructure with predictable performance and responsiveness at scale, we have developed a new multi-chain protocol designed to take advantage of modern cloud infrastructures:

(1) In Linera, a validator is an elastic Web2-like service that validates and executes blocks of transactions in many chains in parallel. Because the number of chains (active and inactive) present in a Linera system is meant to be unlimited, we also call them *microchains*.

(2) The task of actively extending a microchain with new blocks is separate from validation or execution and is assumed by the *owner(s)* of each chain. Every Linera user is encouraged to create a chain that they own and place their accounts there.

(3) Every validator manages all the microchains. (We call this the *integrated multi-chain* approach.) Microchains interact using asynchronous messages and otherwise run independently. As a result, validators can scale elastically by dividing their workload between many internal workers (aka *shards*). Asynchronous messages between chains are implemented efficiently using the internal network of each validator.

(4) Microchains may differ in the way they accept new blocks. When extending their own chains, users submit new blocks directly to validators using a low-latency, mempool-free protocol inspired by reliable broadcast [7, 12]. Applications that require more complex interactions between users may also rely on *ephemeral microchains* created on demand. In practice, only the *public microchains* owned by the Linera infrastructure necessitate a full BFT consensus protocol [12].

(5) As a rule, validators do not interact—except for public chains owned by the infrastructure. Synchronization of microchains between validators is delegated to chain owners. This means that inactive microchains (those not creating blocks) have no cost for validators other than storage.

Using elastic validators is a distinctive assumption of Linera. We intend for the Linera community to support a variety of cloud providers that new validators can choose from. Linera was initially inspired by the academic low-latency payment protocol FastPay developed at Meta [7]. Linera generalizes FastPay notably by turning user accounts into microchains, adding smart contracts, and supporting arbitrary asynchronous messages between chains. A more detailed description of the Linera multi-chain protocol is given in Section 2. We analyze the protocol in Section 3.

1.5.2 Making multi-chain programming mainstream

Linera integrates many chains in a unique set of validators. This greatly facilitates cross-chain communication thanks to the internal network of each validator. For the first time, a variety of Web3 applications have the opportunity to scale elastically by taking advantage of a cheap and efficient multi-chain architecture. To promote the adoption of multi-chain programming, we have made the following design choices:

(6) The execution model of Linera is designed to be language-agnostic and developer-friendly. The initial SDK of Linera will be based on Wasm and will target the Rust programming language.

(7) Linera applications are composable and multi-chain. Once an application is created, it can run on demand on any chain. The running instances of the same application coordinate across chains using asynchronous messages and pub/sub channels. Applications that are running in the same microchain interact using cross-contract calls and ephemeral session objects.

Session objects in Linera are inspired by *resources* in the Move language [9]. Statically-typed resources in Move have been proposed to help with composability [25]. In Linera, resource-like composability is achieved using session handles and runtime checks. For instance, to send tokens, a Linera contract will be able to transfer ownership of a temporary session containing the tokens.

In general, building a large community of developers is a major factor in the adoption of blockchain infrastructures. Because the Wasm ecosystem is continuously improving its multi-language tooling [4], it offers the long-term possibility for Linera to serve several developer communities. See Section 4 for a more detailed discussion of the programming model of Linera.

1.5.3 Robust decentralization for elastic validators

The classical "blockchain trilemma" [10] asserts the difficulty of simultaneously achieving scalability, security, and decentralization. While this observation certainly holds for validators of fixed capacity, we believe that insufficient efforts have been made in the definition and implementation of a satisfying notion of decentralization for elastic validators.

(8) Linera relies on delegated proof of stake (DPoS) for security and supports regularly changing sets of validators. Thanks to the chaining of blocks, the past transactions, the cross-chain messages, and the execution state of each microchain are tamper-resistant.

(9) Microchains are designed to be auditable independently. This means that Linera as a whole will be auditable in a distributed way by the community, using only commodity hardware.

Using large validators for performance and maintaining decentralization using community-driven auditors has been discussed by the blockchain community in the context of rollups [10]. As the Linera project makes progress, we will continue to monitor the technical advances in the field of validity ("ZK") proofs and chain compression. Decentralization of Linera is further discussed in Section 5.

2 The Linera Multi-Chain Protocol

We now introduce the multi-chain protocol at the core of the Linera infrastructure. This technical description is meant to illustrate the main ideas of the protocol without being exhaustive. We analyze the protocol informally in Section 3 and discuss the programming model in Section 4.

2.1 Participants: users, validators, chain owners

The Linera protocol aims to provide a computing infrastructure where developers create decentralized applications and end users interact with them in a secure and efficient way.

As usual for blockchain systems, the state of an application in Linera is replicated across several partially-trusted nodes called *validators*. Modifying the state of an application is done by inserting a *transaction* into a new block and submitting the new block to the validators.

To support scalability requirements, Linera is designed from the start as an integrated multi-chain system: instead of using a single chain, transactions are organized in many parallel chains of blocks, called *microchains*. This means that the state of Linera applications is typically distributed across chains. Importantly, unless a reconfiguration is in progress (Section 2.9), a single set of validators is in use for all the microchains.

In Linera, the task of extending a chain with new blocks is separate from the task of validating blocks. Proposing blocks is assumed by the *owners* of a microchain. In practice, the owner(s) of a chain can be any participant to the protocol. Because Linera validators act as a block validation service, chain owners may also be referred to as *clients*. Examples of chain owners include:

- End users who wish more control over their accounts in different applications;
- End users who wish to operate a temporary chain (*e.g.* for an atomic swap);
- Developers who wish to publish code or manage applications;
- Validators who collectively run a *public* chain (*e.g.* for infrastructure purposes).

The last use case is how Linera manages the current set of validators, also known as the *committee*. The programming model of Linera is presented in Section 4. The additional role of *auditors* is discussed in Section 5.

2.2 Security model

Linera is designed to be *Byzantine-Fault Tolerant (BFT)* [13]. All participants generate a key pair consisting of a private signature key and the corresponding public verification key. Linera uses a delegated proof of stake (DPoS) model [28], where the voting power of each validator is bound to its stake and the stake delegated to it by users.

Assumptions. We present the Linera protocol for a total voting power of N. A fixed, unknown subset of *Byzantine* (aka *dishonest*) validators may deviate from the protocol. It is assumed that they control at most f voting power for some value f such that $0 \leq f < \frac{N}{3}$. This is similar to many BFT protocols [7, 13]. In practice, one chooses the largest possible value for f, namely $f = \lfloor \frac{N-1}{3} \rfloor$.

We do not make any assumptions about users, chain owners, or on the networking layer when it comes to safety properties. Unless specified otherwise, liveness properties do not depend on network delays or message ordering. In other words, the network is *asynchronous* [13].

We use the word *quorum* to refer to a set of signatures issued by validators with a combined voting power of at least $N - f$. An important property of quorums, called *quorum intersection*, is that for any two quorums, there exists an honest validator α that is present in both. When data (typically a block) is signed by a quorum of validators, it is said to be *certified*. Certified data is also called a *certificate* for short.

Goals. Linera aims to guarantee the following security properties:

- *Safety:* For any microchain, every validator sees (a prefix of) the same chain of blocks, therefore it applies the same sequence of modifications to the execution state of the chain and eventually delivers the same set of messages to the other chains.
- *Eventual consistency of chains:* If a microchain is extended with a new certified block on an honest validator, any user can take a series of steps to ensure that this block is added to the chain on every honest validator.

- *Eventual consistency of asynchronous messages:* If a microchain receives a cross-chain message on an honest validator, any user can take a series of steps to ensure that this message is received by the chain on every honest validator.
- *Authenticity:* Only the owner(s) of a microchain can extend their microchain.
- *Piecewise Auditability:* There is sufficient public cryptographic evidence for the state of Linera to be audited for correctness in a distributed way, one chain at a time.

For single-owner chains (Section 2.4), Linera also guarantees the following properties:

- *Monotonic block validation:* In a single-owner chain, if a block proposal is the first one to be signed by the owner at a given block height and it is accepted by an honest validator, then with appropriate actions, the chain owner always eventually succeeds in gathering enough votes to produce a certificate.
- *Worst-case Efficiency:* In a single-owner chain, Byzantine validators cannot significantly delay block proposals and block confirmations by correct users.

2.3 Notations

We assume a collision-resistant hash function, noted $\mathsf{hash}(\cdot)$, as well as a secure public-key signature scheme $\mathsf{sign}[.]$. A quorum of signatures on a block B forms a certificate noted $C = \mathsf{cert}[B]$. In the rest of this report, we identify certificates on the same block B and simply write $C = \mathsf{cert}[B]$ when C is any certificate on B.

The state of the Linera system is replicated across all validators. For a given validator, noted α, we use the notation $X(\alpha)$ to denote the current view of α regarding some replicated data X. A *data type* $D = \langle \mathsf{Tag}, \arg_1, \ldots, \arg_n \rangle$ is a sequence of values starting with a distinct marker Tag and meant to be sent over the network. We use capitalized names to distinguish data type markers from mathematical functions (e.g. hash) or data fields (e.g. $\mathsf{owner}^{id}(\alpha)$), and simply write $\mathsf{Tag}(\arg_1, \ldots, \arg_n)$ for a data type. We write $\tilde{D}$ for a sequence of data types $(D_1, \ldots D_n)$.

2.4 Microchains

The main building blocks of the Linera infrastructure are its microchains. A microchain (or simply *chain* for short) is similar to a regular blockchain in the sense that it is made of a chain of blocks, each containing a sequence of transactions. Importantly, Linera separates the role of proposing new blocks (chain owners' role) from validating them (validators' role). The protocol to extend a chain is configurable and depends on the *type* of the chain.

Chain identifiers. A microchain is represented by an identifier *id* designed to be non-replayable. Specifically, a *unique identifier* (or simply *identifier*) is a non-empty sequence of numbers written as $id = [n_1, \ldots, n_k]$ for some $1 \leq k \leq k_{\mathsf{max}}$. We use :: to denote the concatenation of one number at the end of a sequence: $[n_1, \ldots, n_{k+1}] = [n_1, \ldots, n_k] :: n_{k+1}$ ($k < k_{\mathsf{MAX}}$). In this example, we say that $id = [n_1, \ldots, n_k]$ is the *parent* of $id :: n_{k+1}$.

A Linera system starts with a fixed set of microchains defined in the genesis configuration. To create a new chain, the owner of an existing chain must execute a chain-creation transaction. The new identifier is computed as the concatenation of the parent identifier and the index of the transaction creating the new chain.

Chain types. Linera supports three types of microchains:

(i) *Single-owner chains* where only one user (as identified by its public key) is authorized to propose blocks;

(ii) *Permissioned chains* where only a well-defined set of cooperating users are authorized to propose blocks;

(iii) *Public chains* where validators propose blocks.

In all three cases, the agreement between validators regarding the next block B of a chain is represented in fine by a certificate $C = \mathsf{cert}[B]$. In the case of a single-owner chain, the production of the certificate C is inspired by reliable broadcast [7,12] and will be described in detail in Section 2.8. In the case of public chains, the certificate C is a proof of commit produced by a classical BFT consensus protocol between validators. The case of permissioned chains and public chains is sketched in Section 2.9. For simplicity, unless mentioned otherwise, the rest of this report focuses on single-owner chains.

Every chain includes a field $\mathsf{owner}^{id}(\alpha)$ to authenticate their *owner(s)*, if any. We write $\mathsf{owner}^{id}(\alpha) = pk$ when the chain has a single owner authenticated by the public-key pk. Permissioned chains have $\mathsf{owner}^{id}(\alpha) = \{pk_1, \ldots, pk_n\}$ and public chains $\mathsf{owner}^{id}(\alpha) = \star$. When $\mathsf{owner}^{id}(\alpha) = \bot$, the chain is said to be *inactive*.

Chain lifecycle. Any existing chain can create a new microchain for another user and use the block certificate C as a proof of creation. Once created, the new microchain works independent from its parent microchain. Linera will make available a dedicated public chain to allow new users to easily create their first chain.

Linera also makes it possible to safely and verifiably transfer the control of a chain to another user by executing a transaction that changes the key $\mathsf{owner}^{id}(\alpha)$. Setting $\mathsf{owner}^{id}(\alpha) = \bot$ effectively deactivates the chain permanently.

Using unique identifiers is important so that the state of a deactivated microchain can be safely deleted and archived in cold storage while preventing the chain of blocks from being replayed.

Blocks. A *block* is a data type $B = \mathsf{Block}(id, n, h, \tilde{T})$ made of the following data:

- the unique identifier of the chain to extend id,
- a block height $n \geq 0$,
- the hash h of the previous block (or $\bot$ if $n = 0$),
- a sequence of *transactions* $\tilde{T}$.

A transaction T is an instruction meant to be executed on a chain. Transactions are typically used to modify the *execution state* of the chain. In Linera, they may also have additional effects such as creating chains, sending messages to a *recipient* chain id', or receiving messages.

A microchain id with a current chain of blocks $\bot \to B_0 \to \ldots \to B_n$ is successfully extended by block B when validators receive a certified request $C = \mathsf{cert}[B]$ that contains id and the next expected block height $n+1$. Validators track the current state of each chain id and only vote in favor of adding a block B after validating the correct chaining and the correct execution of B. Under BFT assumption, this ensures that validators eventually execute the same sequence of blocks on each chain, therefore agree on the execution state.

The execution of a block B consists in interpreting the transactions $\tilde{T}$ listed in B in the given order. Transactions may produce outgoing messages for other chains and consume incoming messages. In practice, for auditing purposes, blocks B also include the hash of the state after executing the block, as well as the outgoing messages produced by transactions.

2.5 Cross-chain requests

The state of a Linera application is usually distributed across many chains for scalability. To coordinate across chains, applications rely on asynchronous communication (see also Section 4.3 on programmability).

At the protocol level, asynchronous communication between chains relies on an important mechanism called *cross-chain requests*. Concretely, the execution of a transaction in a block on a chain id by a validator α may sometimes trigger a one-time, asynchronous interaction that will modify the state of another chain id'. (See Algorithm 1 for an example of pseudo-code with cross-chain requests.) Cross-chain requests are cheaply implemented using remote procedure calls (RPCs) in the internal network of each validator: the implementation needs only ensure that each request is executed exactly once.

Importantly, arbitrarily modifying the execution state of a target chain with a cross-chain request is not possible in general because validators do not agree on the order of execution of cross-chain requests—in other words, this would break the *Safety* property. While FastPay [7] uses cross-chain requests for payments only, Linera uses this mechanism to create new chains and to deliver messages to the *inbox* of an existing chain.

Inboxes allow Linera to support arbitrary messages because the modification is not applied to the target chain immediately. Rather, the message is placed in the target chain's inbox, implemented as a commutative data structure (*i.e.* where the order of insertions does not matter) described in Section 2.6. The owner(s) of the receiving chain then executes a transaction that picks the message from the inbox and applies its effect to the chain state (Section 2.7).

2.6 Chain states

We now describe the state of the Linera chains as seen by validators and clients. Every validator stores a map that contains the states of all the chains, indexed by their identifiers. Clients have a similar representation of the chains except that they act as a *full-node* (*i.e.* track the chain of blocks and execution state) only for a small subset of the chains relevant to them. Next, we focus on the state of a given validator, noted α.

Chain state. The state of a chain id as seen by a validator α can be divided into (i) a *consistent part* which is a deterministic function of the chain of blocks $\perp \to B_0 \to \ldots \to B_n$ already executed by α; and (ii) a *localized* part on which validators may not agree. The consistent part of a chain state includes the following data:

- A field $\mathsf{owner}^{id}(\alpha)$ controlling the production of blocks in id, as seen before.
- An integer value, written $\mathsf{next\text{-}height}^{id}(\alpha)$, tracking the expected block height for the next block of id. (Here $n + 1$. Initially 0.)
- The hash of the previous block $\mathsf{block\text{-}hash}^{id}(\alpha)$ (initially $\perp$.)
- The execution state, noted $\mathsf{state}^{id}(\alpha)$.

The localized part of a chain state includes the following:

- $\mathsf{pending}^{id}(\alpha)$, an optional value indicating that a block on id is pending confirmation (the initial value being $\perp$).
- A list of certificates, written $\mathsf{received}^{id}(\alpha)$, tracking all the certificates that have been confirmed by α and involving id as a recipient chain.
- A data-structure called an *inbox* and denoted by $\mathsf{inbox}^{id}(\alpha)$ (see next paragraph).

The field $\mathsf{pending}^{id}(\alpha)$ is specific to single-owner chains and explained in Section 2.8. It is completed by additional data in the case of permissioned and public chains. The list of certificates $\mathsf{received}^{id}(\alpha)$ is crucial for liveness (Section 3.3).

Inbox state. An inbox $I = \mathsf{inbox}^{id}(\alpha)$ is a special data structure used to track the cross-chain messages received by id and waiting to be consumed by a transaction. Specifically, messages are *added* to an inbox upon reception and *removed* from it after being executed by the receiving chain.

An important property of an inbox is that adding or consuming distinct messages is commutative. In the simplest implementation, one can think of an inbox as two disjoint sets of messages $I = (I_+, I_-)$. We may define the addition of a message m to I, noted $I + m$, as $(I_+ \cup \{m\}, I_-)$ if $m \notin I_-$ and $(I_+, I_- \setminus \{m\})$ otherwise. Similarly, the subtraction $I - m$ is $(I_+, I_- \cup \{m\})$ if $m \notin I_+$ and $(I_+ \setminus \{m\}, I_-)$ otherwise. In this setting, when $\mathsf{inbox}^{id}(\alpha) = (I_+, I_-)$, the set I_+ represents the messages m that have been received by id and are waiting to be executed in a next block; I_- tracks the messages that have not been received by id yet (from the point of view of α) but were nonetheless executed by anticipation because of a certified block. In this simplified presentation, we are assuming that messages are never replayed identically, say, because they include a counter for each pair of sender and receiver (id, id').

The current implementation of Linera uses a more complex data structure enforcing an ordered delivery of messages for each pair of sender and receiver, and for each application. See Appendix A.1 for a detailed description. For simplicity, in what follows, we still use the notation inbox^{id}_{-} to denote the equivalent of the set I_- above, representing the executed messages waiting to be received by the chain id at a given moment.

2.7 Block execution

We now describe how to execute the sequence of transactions contained in a chain of blocks. The transactions T supported by a Linera deployment include the following commands:

- $\mathsf{OpenChain}(id', pk')$ to activate a new chain with a fresh identifier id' and public key pk'—possibly on behalf of another user who owns pk';
- $\mathsf{ChangeKey}(pk')$ to transfer the ownership of a chain;
- $\mathsf{CloseChain}$ to deactivate the chain id;
- $\mathsf{Execute}(o)$ to execute a *user operation* o;
- $\mathsf{Receive}(m)$ to pick a *cross-chain message* m from the chain inbox and execute it.

The first three types of transactions are examples of *system operations* that are predefined in the protocol. In constrast, user operations o are executed by user-defined applications (aka "smart contracts"). At a high level, operations are meant to be freely added by the producer of a block, whereas receiving a cross-chain message requires the message to be first sent by another transaction of another chain (2.5).

For simplicity, we have omitted transaction fees and additional logic required by multi-owner chains and reconfigurations (Section 2.9). Formally, to execute user operations o, we assume a method $\textsc{ExecuteOperation}(id, o)$ that attempts to modify state^{id} and may return either $\perp$ or (m, id') in case of success, the latter case being a request that a message m be sent to the chain id'. We also assume a method to modify state^{id} by executing a cross-chain message m, noted $\textsc{ExecuteMessage}(id, m)$. Importantly, receiving a message m may produce another message m' in return.

This description translates to the pseudo-code in Algorithm 1. The execution of a block $B = \mathsf{Block}(id, n, h, \tilde{T})$ as suggested above corresponds to the function ExecuteBlock. The validation of blocks by the function BlockIsValid is similar to ExecuteBlock except that no change to the state is persisted, cross-chain queries are ignored, and messages cannot be executed by anticipation, that is, the validation fails if inbox_{-}^{id} is not empty at the end of the call.

2.8 Client/validator interactions

We can now describe the interactions between clients (aka chain owners) and validators in a Linera system. Clients to the Linera protocol run a local node, noted β, that tracks a small subset of chains relevant to them. These relevant chains typically include the ones owned by the client as well as direct dependencies, notably a special Admin chain in charge of tracking validators and their networking addresses (Section 2.9).

Network interactions with validators are always initiated by a client. Clients may wish to either (i) extend one of their own chain(s) with a new block, or (ii) provide a *lagging* validator with the certificates that it is missing in a chain of interest to the client.

To support these two use cases, validators provide two service handlers described in Algorithm 2 and called HandleRequest and HandleCertificate. For simplicity, we omit the service handlers used by clients to query the state of a chain or to download a chain of blocks from a validator.

We start with the interactions meant to update a lagging validator.

Uploading missing certificates to a validator. Any client may upload a new certificate $C = \mathsf{cert}[B]$ with $B = \mathsf{Block}(id, n, h, \tilde{T})$ to a validator α using the HandleCertificate entry point, provided that the chain id is active and that n is the next expected block height from the point of view of α (*i.e.* formally $\mathsf{owner}^{id}(\alpha) \neq \perp$ and $\mathsf{next\text{-}height}^{id}(\alpha) = n$).

If the validator α has not created the chain id yet or if it is lagging by more than one block, concretely the client should upload a sequence of multiple missing certificates ending with $C = \mathsf{cert}[B]$. If necessary, the sequence may start with blocks of an *ancestor* chain id' (that is, $id' = \mathsf{parent}(\mathsf{parent}(\dots id)))$. In this case, the sequence continues until the block of the parent chain that created the chain id is reached, then finishes with the chain of blocks ending with C.

In practice, the need to upload such a sequence of certificates justifies that the local node β may track the chain id in the first place. The client can quickly find the exact block that created id by looking at the first block logged in the list $\mathsf{received}^{id}(\beta)$.

Extending a single-owner chain. In the common scenario where validators are sufficiently up-to-date, Linera clients may extend their chain with a new block B using a variant of reliable broadcast [7, 12] illustrated in Figure 1 and going as follows.

- The client broadcasts the block B authenticated by its signature to each validator using the HandleRequest entry point α (❶) and waits for a quorum of responses.
- A validator responds to a *valid* request $R = \mathsf{auth}[B]$ of the expected height by sending back a signature on B, called a *vote*, as acknowledgment (❸). After receiving votes from a quorum of validators, a client forms a certificate $C = \mathsf{cert}[B]$.
- When a certificate $C = \mathsf{cert}[B]$ with the expected next block height is uploaded (❹), this triggers the one-time execution of the block B (❺).

A synchronization step is occasionally needed first (⓿) if some validator α is unable to vote right away for an otherwise-valid proposal $B = \mathsf{Block}(id, n, h, \tilde{T})$. This may happen for two reasons:

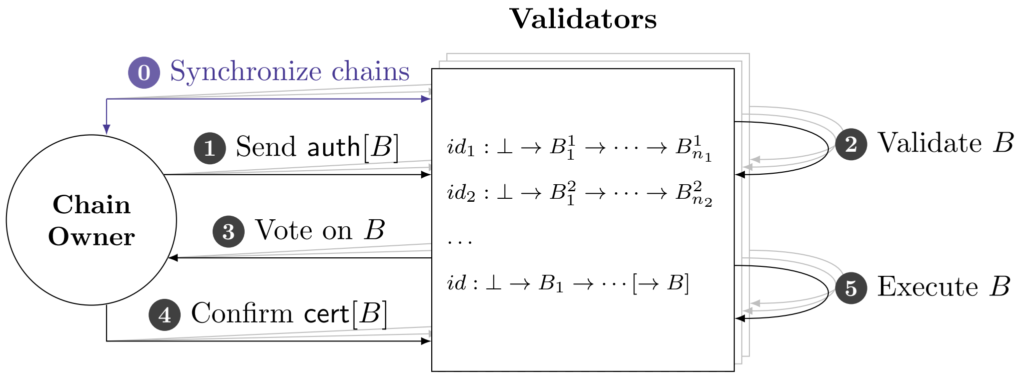
Validators
0 Synchronize chains
1 Send auth[B]
2 Validate B
3 Vote on B
4 Confirm cert[B]
5 Execute B
Chain Owner
$id_1 : \perp \to B^1_1 \to \cdots \to B^1_{n_1}$
$id_2 : \perp \to B^2_1 \to \cdots \to B^2_{n_2}$
$\dots$
$id : \perp \to B_1 \to \cdots [\to B]$

Figure 1: Extending a single-owner chain id with a new block

1. either the chain id is not active yet or α is missing earlier blocks (*i.e.* formally $\mathsf{owner}^{id}(\alpha) = \perp$ or $\mathsf{next\text{-}height}^{id}(\alpha) < n$);

2. α is missing cross-chain messages, that is: $I_{-} = \mathsf{inbox}^{id}_{-}$ is not empty at the end of the staged execution of $\tilde{T}$.

In the first case, the Linera client must upload missing certificates in the chain id (and possibly its ancestors) as described in the previous paragraph, until $\mathsf{next\text{-}height}^{id}(\alpha) = n$. In the second case, the client must upload missing certificates in the chains that have sent the messages $m \in I_{-}$ to id. When B has been correctly constructed (*i.e.* is not trying to receive messages that were never sent), the set I_{-} is necessarily covered by the certificates listed in the union $\bigcup_{\alpha'} \mathsf{received}^{id}(\alpha')$ where α' ranges over any quorum of validators.

Importantly, uploading a missing block to a validator benefits all clients. To maximize liveness and decrease the latency of their future transactions, in practice, it is expected that users proactively update all the validators when it comes to their own chains, therefore minimizing the need for synchronization by other clients. However, the possibility of synchronization by everyone is important for liveness (Section 3.3). It also allows a certificate to act as a proof of finality for the certified block.

In practice, a client should execute the optional synchronization step (0) and the voting step (1) on a separate thread for each validator. To prevent denial-of-service attacks from malicious validators, a client may stop synchronizing validators as soon as enough votes are collected (2).

The steps 1 2 3 used to decide a new block in a single-owner chain constitute a 1.5 round-trip protocol. Inspired by reliable broadcast, this protocol does not have a notion of “view change” [12] to support retries. In other words, a chain owner that has started submitting a (valid) block proposal B cannot interrupt the process to propose a different block once some validators have voted for B. Doing so would risk blocking their chain. For this reason, Linera also supports a variant with an extra round trip (Section 2.9).

2.9 Extensions to the core protocol

We now sketch a number of important extensions to the core Linera multi-chain protocol.

Permissioned chains. The protocol presented in Section 2.8 allows extending a single-owner microchain optimistically in 1.5 client-validator round trips. Linera also supports a more complex protocol with 2.5 round trips to address the following use cases:

- A single chain owner wants to be able to safely interrupt ongoing block proposals while they are in progress.

- Transactions in blocks depend on external oracles (*e.g.* Unix time) and include conditions that may become invalid after being valid.
- Multiple owners wish to operate the chain (assuming minimal off-chain coordination).
- A single chain owner wishes to delegate maintenance operations related to validator reconfigurations.

We omit the details of the 2.5 round-trip protocol for brevity. It can be seen as a simplified partially-synchronous BFT consensus protocol [12] with view changes (aka rounds) but without leader election or timeouts. In the absence of leader election, different owners may try to propose a different block at the same time (*i.e.* in the same block height and round) causing the current round to fail and another round to be needed. As a consequence, this mode of operation assumes that the owner(s) of a same chain maintain a sufficient level of (off-chain) cooperation so that ultimately only one of them proposes a block and succeeds.

Public chains. Public chains are used in the remaining use cases: when a chain continuously produces new blocks with the help of validators. In this case, the transactions authorized in a block are likely to be only those receiving cross-chain messages from other chains. Examples of applications include:

- Managing validators and stakes in one place (see reconfigurations below).
- Running traditional blockchain algorithms (e.g. AMMs) that were not designed to take advantage of the multi-chain approach;
- Facilitating the creation of microchains for new users.

Public chains in Linera will be based on a full BFT consensus protocol. This is the only case in the Linera infrastructure where Linera validators take an active role in block proposals. We plan to rely on user chains and cross-chain messages instead of a traditional mempool to gather user transactions into new blocks.

Pub/sub channels. A common use case for cross-chain asynchronous messages is for an application instance on a chain *id* to create a *channel* and maintain a list of *subscribers* to it. Specifically, a channel operates as follows:

- Transactions executed on the chain *id* may push new messages to the channel;
- When this happens, the current subscribers receive a cross-chain message in their inbox;
- The set of subscribers is managed on the chain *id* by receiving and executing messages $\mathsf{Subscribe}(id')$ and $\mathsf{Unsubscribe}(id')$ from subscribers id'.

We have found pub/sub channels to be a useful abstraction when programming Linera applications (see also Section 4). The Linera protocol supports pub/sub channels natively in order to enable specific optimizations. For instance, newly accepted subscribers currently receive the last message of a channel without additional work from the owner of the channel.

Reconfigurations. Being able to change the set of Linera validators (aka the *committee*) is crucial for the security of the system (see Section 5).

To do so, Linera deploys a dedicated ADMIN public chain running the application for system management. This system application is in charge of keeping track of the successive sets of validators, aka *committees*, including their stakes and network addresses. The successive configurations produced by this application are identified by their *epoch* number.

To safely disseminate the information that the set of validators is changing, the ADMIN publishes new configurations to a special channel that every Linera microchain is subscribed to when created.[2] A newly created microchain automatically receives the current validator set (*i.e.* the last message in the admin channel) and sets its current epoch number field.

When a new committee is created, every microchain receives a message in its inbox. Importantly, microchain owners must include the incoming message in a new block to explicitly migrate their chain to the new set of validators. This must be done when both sets of validators are still operating, before the previous set stops.

Thanks to the scalable nature of Linera, migrating a large number of chains to a new configuration in a short period of time is doable in parallel provided that enough clients are active. To facilitate this process and allow chain owners to go offline for an extended period, we envision that many users will authorize a third party to create the migration blocks on their behalf. This will however require configuring the chain to use the 2.5 round-trip protocol mentioned above for the duration of the authorization.

To prevent long-range attacks, the ADMIN chain will also regularly suggest old committees to be *deprecated*. After accepting such an update, microchains will ignore messages in blocks certified only by deprecated committees. The old messages will be accepted again only after they are included in a chain of blocks ending with a trusted configuration (hence *re-certified*).

3 Analysis of the Multi-Chain Protocol

In this section, we analyze the design goals set by the Linera blockchain, including responsiveness, scalability and security guarantees.

3.1 Responsiveness

A common problem when interacting with classical blockchains is the lack of performance guarantees. Transactions submitted to the mempool may be picked instantly, after a moment, or never, depending on the other user transactions posted around the same time. Canceling a pending transaction typically requires submitting another one with a higher gas fee. Furthermore, classical blockchains have a fixed and limited throughput: large enough bursts of submitted transactions (*e.g.* due to a popular airdrop) must eventually cause a backlog and/or a surge in transaction fees. Mempool systems also expose users to value drainage with Miner Extractable Value (MEV) techniques.

Linera allows users to manage their own chain and work around these problems thanks to a lightweight block extension protocol inspired by client-based reliable broadcast (Section 2.8). This approach does not require a mempool, as users submit their transactions directly to the validators and fully control the processing time. The parallel communication with the validators means that the only processing delay is imposed by the network round-trip time (RTT) between the client and the validators (usually a few hundred milliseconds). Finally, we anticipate that removing the mempool and diminishing latency should greatly reduce MEV opportunities.

3.2 Scalability

The microchain approach (Section 2.4) allows Linera validators to be efficiently sharded across multiple workers. Concretely, each worker in a validator is responsible for a particular

[2]Alternatively, a tree of public chains that relay the stream of configurations may be considered in the future for scalability purposes.

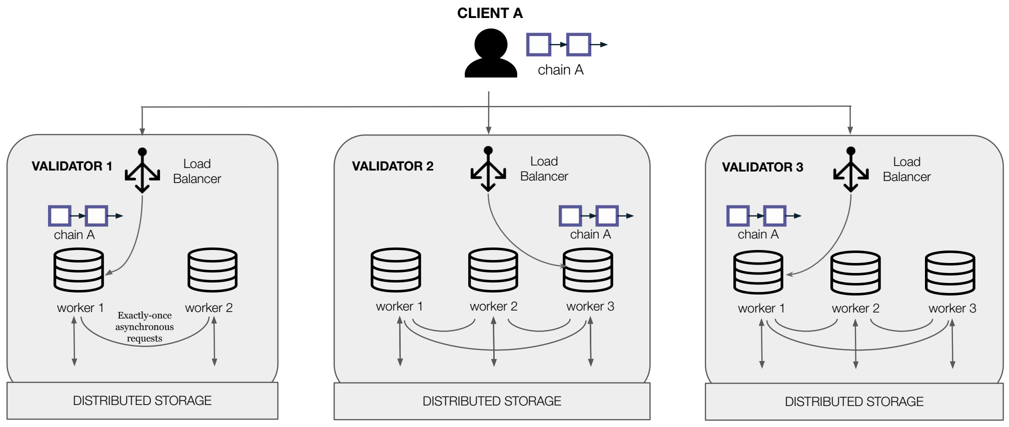

Figure 2: Architecture of Linera validators.

subset of microchains. Clients communicate with the load balancer of each validator, which dispatches queries internally to the appropriate worker (Figure 2).

This design allows Linera to scale horizontally as the load of the system increases: each validator only needs to add worker machines to cope with the traffic. Importantly, sharding is internal: the number of workers and the assignment of microchains to workers do not need to be consistent across validators.

Workers within a single validator belong to a single entity and thus trust one another. This makes the communication between workers—and therefore the cross-chain requests of Linera (Section 2.5)—quick and inexpensive.

The sharding model of Linera is different from the approach called *blockchain sharding* [31, 33]. In the latter, cross-chain messages are exchanged between groups of mutually distrusting nodes (i.e., the validators in charge of each shard) usually spread across the Internet. This incurs significant overhead. Linera uses point-to-point communication across co-located workers that trust each other and requires much fewer resources. At the same time, larger validators can be efficiently audited by clients who want to control their operations. We describe the audit operation in Section 5.

The elastic architecture of Linera allows validators to adapt to traffic fluctuations. When an increased number of transactions are submitted, it is easy to increase the number of cloud-based workers processing the transactions. The same workers can be quickly turned off when no longer needed to reduce costs.

The *public chains* of Linera require a full BFT consensus protocol to order blocks submitted by multiple clients (Section 2.9). Yet, the consensus protocol is instantiated once per public microchain rather than once for the entire system. This has a number of benefits. First, users from different public microchains cannot degrade each other's experience. Second, the transaction rate of a single microchain is not a limiting factor for the entire system. Ultimately, the throughput of Linera can be always increased by creating additional microchains and augmenting the size of validators.

3.3 Security

In this section, we provide an informal security analysis of the Linera multi-chain protocol. Following the description in Section 2, we focus on single-owner chains. The analysis will be extended to other types of accounts (Section 2.9) in future reports.

Claim 1 (Safety)**.** *For any microchain, every validator sees (a prefix of) the same chain of* *blocks, therefore it applies the same sequence of modifications to the execution state of the chain and eventually delivers the same set of messages to the other chains.*

Indeed, per Algorithm 2, each honest validator votes for at most one valid block at a given height per microchain. By the quorum intersection property (Section 2.2), under BFT assumption, there can be only one block per height per chain certified by a quorum of validators. The set of outgoing messages from a chain (the cross-requests in Algorithm 1) is a deterministic function of the current chain of blocks.

Importantly, asynchronous cross-chain messages are delivered exactly once after they are scheduled. This allows applications to safely transfer assets.

Claim 2 (Eventual consistency of chains)**.** *If a microchain is extended with a new certified block on an honest validator, any user can take a series of steps to ensure that this block is added to the chain on every honest validator.*

Indeed, any user can retrieve the new certificate and its predecessors from the honest validator and deliver it to validators that still have not received it. The exact sequencing in which blocks can be uploaded to a validator is discussed in Section 2.8.

Claim 3 (Eventual consistency of asynchronous messages)**.** *If a microchain receives a cross-chain message on an honest validator, any user can take a series of steps to ensure that this message is received by the chain on every honest validator.*

An asynchronous message is received by a chain on a particular validator only after a block containing a transaction that triggers the message is signed by a quorum and added to the sender's chain. When this happens, the state of the receiving chain is updated to track the origin of the message (see $\mathsf{received}^{id}(\alpha)$ in Section 2.6). This allows a client to download the corresponding block from the same validator if needed. Any honest validator adding the same block for the first time will add the same message to the recipient's inbox.

Claim 4 (Authenticity)**.** *Only the owner(s) of a microchain can extend their microchain.*

Honest validators only accept block proposals if they are authenticated by an owner (Algorithm 2). This ensures that no one else can add blocks to the microchain. Other types of microchains (Section 2.9) implement similar verifications.

Claim 5 (Piecewise Auditability)**.** *There is sufficient public cryptographic evidence for the state of Linera to be audited for correctness in a distributed way, one chain at a time.*

Any Linera client can request a copy of any microchain and re-execute the certified blocks. This allows verifying the successive execution states and the set of outgoing messages from the chain. Execution states are typically compared across validators by including execution hashes in blocks. The received messages of a chain should be compared to the outgoing messages from the other chains (Section 5.2).

Claim 6 (Worst-case Efficiency)**.** *In a single-owner chain, Byzantine validators cannot significantly delay block proposals and block confirmations by correct users.*

Linera clients contact all the validators in parallel and consider an operation as completed as soon as they receive signatures from a quorum of validators (Section 2.8).

Claim 7 (Monotonic block validation)**.** *In a single-owner chain, if a block proposal is the first one to be signed by the owner at a given block height and it is accepted by an honest validator, then with appropriate actions, the chain owner always eventually succeeds in gathering enough votes to produce a certificate.*

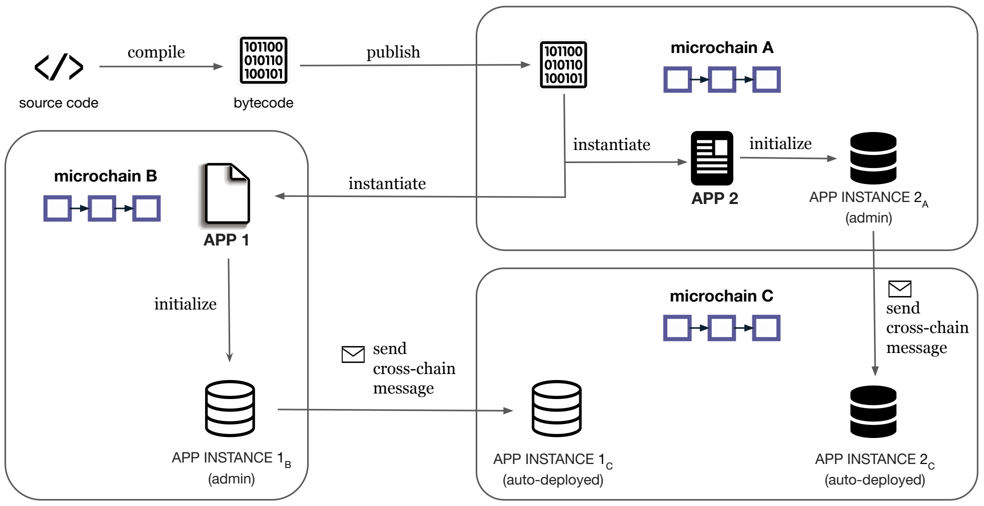

Figure 3: Linera applications.

If the block proposal B for a chain id is accepted by a validator and is the first one ever signed at this height, this means that every other validator α has already accepted the proposal (*i.e.* $\mathsf{pending}^{id}(\alpha) = B$) or has not voted yet (*i.e.* $\mathsf{pending}^{id}(\alpha) = \perp$). In the latter case, block validation may temporarily fail for α if some earlier blocks or messages are missing: this can be resolved by updating the validator with the missing blocks (see Section 2.8). After proper synchronization, in the absence of external oracle and non-deterministic behaviors, submitting the proposal B to the validator will eventually produce the expected vote for B.

4 Building Web3 Applications in Linera

The programming model of Linera [1] is designed to provide rich, language-agnostic composability to application developers while taking advantage of microchains for scaling.

4.1 Creating applications

Linera uses the WebAssembly (Wasm) virtual machine [3, 23] as the execution engine for user applications. The SDK to develop Linera applications will be initially targeting the Rust language.

An application is created in several steps (Figure 3). First, a software module (aka smart contract) in Rust is compiled to Wasm bytecode. The bytecode is then published by its author on a microchain of its choice and receives a unique *bytecode identifier*. Next, the bytecode is instantiated using the bytecode identifier and specific application parameters (*e.g.* name of the token, token supply, etc). This operation creates a fresh *application identifier* ("APP 1" in Figure 3) and initializes the local state of the application ("APP INSTANCE 1_B"). This initial local state may hold specific parameters to help administrating the new application in the future.

A single bytecode identifier can spawn across multiple, independent applications that share the same code but do not share the same configuration ("APP 1" and "APP 2" in Figure 3).

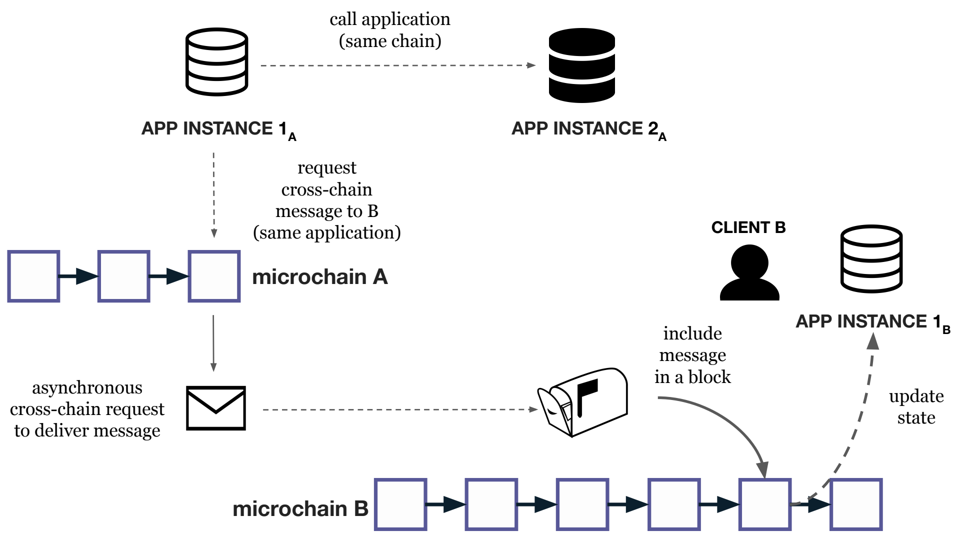

Figure 4: Cross-chain messages (left and bottom) vs. cross-contract calls (top).

4.2 Multi-chain deployment

Linera applications are multi-chain by default in the sense that their global state is generally split across several chains. In other words, the local instance of an application at a given chain holds only the subset of the application state that is located there. For instance, in an ERC-20-like token management application, the owner of a single-owner chain may want to hold their personal accounts on the chain that they own.

The bytecode of an application is automatically downloaded and the application started when the owner of a microchain accepts an incoming message (Section 2.5) from the application for the first time ("APP INSTANCE 1_C" in Figure 3).

4.3 Cross-chain communication

Cross-chain communication between applications is realized using asynchronous calls to allow microchains to run independently. The programming style for cross-chain coordination between Linera applications is inspired by the actor model [6]. The implementation relies on cross-chain requests described in Section 2.5. The fundamental point is that each actor has exclusive access to its own internal state and that actors cannot call each other directly.

Cross-chain messages. Cross-chain messages allow an application to transfer arbitrary data asynchronously from one chain to another (Figure 4). To make sense of the data, the same application must be on the sending end and on the receiving end of a cross-chain message. In practice, the local instance of an application maintains an inbox per origin that the instance has communicated with. When an application wants to send a message to a destination, it returns a value containing the message so that the runtime can execute the appropriate cross-chain request.

Contrary to FastPay [7] and Zef [8], Linera is not limited to payment requests and can deliver arbitrary cross-chain messages defined by user applications. The effects of cross-chain messages do not generally commute, therefore in Linera, the ordering in which incoming messages are received and then executed by a recipient's chain is important. We solve this issue by relying on block proposers to specify the ordering of incoming messages when picking the messages from the chain inboxes.

In general, messages are not guaranteed to be picked on the receiving side. When they are, the current implementation forces messages to be picked in order. This general policy will likely be refined in the future to account for specific use cases, notably for public chains where block production never stops (Section 2.9).

Pub/sub channels. On top of the one-to-one communication, Linera supports one-to-many communication using *channels*. A user can create a channel within an application, while the same application's instances residing on other microchains can subscribe to it by sending a subscribe message with the publisher application and chain identifiers. Importantly, a subscriber is added to a channel only when the publisher accepts the subscription by adding the registration message to its chain. Under the hood, channels act as a set of one-to-one connections. A message sent to a channel is delivered to all the inboxes that are subscribed to the channel and can be picked up by the subscribers. By design, a late subscriber, once accepted by the publisher, receives the last message sent to the channel—rather than the entire history of messages.

4.4 Local composability

Synchronous calls. On the same microchain, different Linera applications can be composed using synchronous calls similar to smart-contract calls in classical blockchains such as Ethereum [32] (see the top part of Figure 4). The state modifications resulting from a sequence of application calls and originating from a single user transaction are atomic. In other words, either all of the calls succeed or all of them fail. Calling an application creates a virtual copy of its internal state and executes the call on the cached state. At this point, the new state is not yet written to storage. If any of the transactions fails, all the staged modifications are discarded.

Sessions. In some cases, it is desirable to delegate the management of a piece of state from one application to another. We call the temporary object managing such a detached state a *session*. A typical example of a use case may go as follows: (i) an application B calls into the token-management application A; (ii) some tokens are withdrawn from the ledger of A and put into a new session; (iii) B receives ownership of the session; (iv) B calls into the session to move the tokens back to the ledger of A, say, under another account; this effectively consumes and terminates the session.

Sessions are guaranteed to be owned by a single application (no duplication). Consuming a session is not optional: sessions must be properly consumed before the end of the current transaction, otherwise, the transaction will fail. In addition to assets, sessions are thus suitable for managing temporary obligations, for instance, the obligation to pay back a flash loan [25].

4.5 User authentication

Applications often need to authenticate end users in order to authorize certain actions. For instance, transferring an asset should require the permission of its owner.

In Linera, users are authenticated when they propose a block in a chain that they own (Section 2.8). During execution, the identity of the user that signed the current block, called the *authenticated signer*, is visible to all the operations contained in the block by default.

An operation creating a cross-chain message may optionally propagate the current authenticated signer along with the message. This is important so that assets temporarily placed on another chain (say, a public chain) may be claimed by their owner.

Similarly, authenticated signers may be propagated when calling another application on the same chain. This allows applications to program new categories of assets and make them available to other applications using abstract APIs.

4.6 Ephemeral chains

Another specificity of the programming model of Linera is the ability to create short-lived permissioned chains (Section 2.9) meant for a short interaction between a small number of loosely coordinated users.

For instance, two users may create a microchain for swapping two assets atomically. The shared microchain will have (up to) two owners and its parameters will be adapted to the exchange process. To use the chain, both users must transfer the assets that they want to exchange from their primary microchains to the shared chain, then one of the users must create a block to confirm or cancel the swap. Importantly, once the swap is concluded, the shared microchain is deactivated. This prevents any further extension of the temporary chain and allows archiving it in the future.

To optimize liveness in the case of an ephemeral permissioned chain (Section 2.9), operations may interact with the user permissions to propose blocks as seen by the consensus protocol. For instance, in the case of a temporary chain for an atomic swap, it is desirable to restrict the ability to propose blocks to those owners who have already locked their assets. Another example is a temporary microchain dedicated to a game of chess between two users. Here, the application can determine which player needs to move and update the microchain consensus layer to accept the next block only from the chosen user. A more realistic chess application may also include a referee as an owner of the temporary chain to enforce progress.

5 Decentralization

Linera encourages validators to use cloud infrastructure to unlock elastic scaling and benefit from standard production environments. To maximize decentralization, Linera relies on two key features: delegated proof of stake (DPoS) and audits by the community.

5.1 Delegated proof of stake

To ensure the long-term security of the system, Linera relies on delegated proof of stake (DPoS): the voting rights of validators are functions of their stakes in the system, together with the stakes that are delegated to them by end users. For DPoS to function correctly, users must be able to change their delegation preferences, and validators must have an automated procedure to join and leave the system. Both operations require a public chain where any user can submit transactions. Reconfiguring validators also requires a carefully-designed migration protocol for every chain. Both mechanisms were sketched in Section 2.9.

Token delegation and economics will be made more precise in a separate document. To address long-range attacks—where old committees become corrupt [17]—, Linera allows microchains to refuse cross-chain messages (*e.g.* payments) from committees that are not trusted anymore (see Section 2.9).

5.2 Auditability

Auditing a blockchain traditionally requires running a *full node* that locally holds a copy of the entire transaction history. However, in the case of a high-throughput system, this may require significant amounts of disk space and CPU resources. When regular users—those using commodity hardware—need days or weeks to fully audit a decentralized system, the community may not be able to credibly deter a coalition of rogue validators from altering the protocol. Light clients [14] reduce resource usage but only check the block headers and do not provide the same level of verification.

In contrast, the microchain approach makes it possible for the community to continuously audit Linera validators. In Linera, an auditor is similar to a client (Section 2.8) in that it only needs to track a small subset of microchains. Because scalability in Linera relies on having many chains rather than larger blocks, it is always feasible to replay the execution of a single chain in real-time on commodity hardware.

For the Linera community to continuously verify all the chains, a distributed protocol can be put in place on top of a shared distributed storage such as IPFS [5] as follows. Executing the blocks in a chain allows to verify the execution state and the outgoing messages. Blocks should typically be marked as audited and the outgoing messages indexed in the distributed storage. To complete the verification of a chain, the client must also verify that each incoming message was indeed produced by its sender chain. This can be done by looking up incoming messages in the shared storage to see if they have been verified already, and otherwise, schedule their verification.

6 Conclusion

Linera aims to deliver the first multi-chain infrastructure with predictable performance, responsiveness, and security at the Internet scale. To do so, Linera introduces the idea of operating many parallel chains, called *microchains*, in the same set of validators, and using the internal network of each validator to quickly deliver the asynchronous messages between chains. This architecture has a number of advantages:

- **Elastic scaling.** In Linera, scalability is obtained by adding chains, not by increasing the size or the rate of blocks. Each validator may add and remove capacity (aka internal workers) at any time to maintain nominal performance for multi-chain applications.

- **Responsiveness.** When microchains are operated by a single user, Linera uses a simplified mempool-free consensus protocol inspired by reliable broadcast [7,12]. This reduces block latency and ultimately makes Web3 applications more responsive.

- **Composability.** Compared to other multi-chain systems, low block latency also helps with composability: it allows receivers of asynchronous messages from another chain to quickly answer by adding a new block.

- **Chain security.** Compared to traditional multi-chain systems, a benefit of running all the microchains in the same set of validators is that creating chains does not impact the security model of Linera.

- **Decentralization.** Linera relies on delegated proof of stake (DPoS) for security. Each microchain can be separately executed on commodity hardware. This allows clients and auditors to continuously run their own verifications and hold validators accountable.

- **Language agnostic.** The programming model of Linera does not depend on a specific programming language. After careful consideration, we have decided to concentrate our efforts on Wasm and Rust for the initial execution layer of Linera.

In future reports, we will formalize the protocols to support multi-owner chains as well as the other extensions mentioned in Section 2.9. In particular, we plan to incorporate a state-of-the-art consensus mechanism (e.g. [16, 22, 27]) on top of our existing multi-chain infrastructure. We also plan to describe the economic models for the fair remuneration of validators and incentivization of users separately. Linera's ability to deactivate and archive microchains provides an elegant venue to control the storage costs of validators in the future. In general, we anticipate that Linera's integrated architecture and the minimization of validator interactions will be extremely helpful when it comes to optimizing the costs of operating validators at scale.

References

[1] Linera developer manual. `https://linera.dev`.

[2] Linera github repository. `https://github.com/linera-io/linera-protocol`.

[3] WebAssembly. `https://webassembly.org/`.

[4] The Bytecode Alliance. `https://bytecodealliance.org/`, 2022.

[5] The InterPlanetary File System. `https://ipfs.tech/`, 2022.

[6] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT press, 1986.

[7] Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance Byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

[8] Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. Zef: Low-latency, scalable, private payments. *arXiv preprint arXiv:2201.05671*, 2022.

[9] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable resources. `https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf`, 2020.

[10] Vitalik Buterin. Endgame. `https://vitalik.ca/general/2021/12/06/endgame.html`, 2021.

[11] Vitalik Buterin. An incomplete guide to rollups. `https://vitalik.ca/general/2021/01/05/rollup.html`, 2021.

[12] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[13] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[14] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. Sok: Blockchain light clients. *Cryptology ePrint Archive*, 2021.

[15] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SSP'20)*, pages 910–927. IEEE, 2020.

[16] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[17] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019.

[18] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. {Bitcoin-NG}: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 45–59, 2016.

[19] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022.

[20] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2016.

[21] Arthur Gervais, Hubert Ritzdorf, Ghassan O Karame, and Srdjan Capkun. Tampering with the delivery of blocks and transactions in bitcoin. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 692–705, 2015.

[22] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Bullshark: DAG BFT protocols made practical. *arXiv preprint arXiv:2201.05677*, 2022.

[23] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[24] Jae-Yun Kim, Junmo Lee, Yeonjae Koo, Sanghyeon Park, and Soo-Mook Moon. Ethanos: efficient bootstrapping for full nodes on account-based blockchain. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 99–113, 2021.

[25] Krešimir Klas. Smart contract development — Move vs. Rust. `https://medium.com/@kklas/smart-contract-development-move-vs-rust-4d8f84754a8f`, 2022.

[26] Michał Król, Onur Ascigil, Sergi Rene, Alberto Sonnino, Mustafa Al-Bassam, and Etienne Rivière. Shard scheduler: object placement and migration in sharded account-based blockchains. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 43–56, 2021.

[27] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[28] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

[29] nanfengpo. A design of decentralized ZK-rollups based on EIP-4844. https://ethresear.ch/t/a-design-of-decentralized-zk-rollups-based-on-eip-4844/12434, 2022.

[30] Slashdot. Best blockchain apis of 2022. https://slashdot.org/software/blockchain-apis/, 2022.

[31] Alberto Sonnino. Chainspace: A sharded smart contract platform. In *Network and Distributed System Security Symposium 2018 (NDSS 2018)*, 2018.

[32] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[33] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

Algorithm 1 Block execution and validation

```
function INIT($id$)                                          ▷ Set up a new chain if necessary
    if $id \notin$ chains then
        owner$^{id} \leftarrow \perp$
        next-height$^{id} \leftarrow 0$
        block-hash$^{id} \leftarrow \perp$
        state$^{id} \leftarrow$ state$^{id}(init)$                               ▷ Genesis state for $id$
        pending$^{id}(\alpha)$
        received$^{id} \leftarrow [\,]$
        inbox$^{id} \leftarrow (\{\}, \{\})$

function EXECUTETRANSACTION($id$, $T$, $C$)
    switch $T$ do
        case OpenChain($id', pk'$):
            ensure $id' = id$ :: next-height$^{id}$
            do asynchronously                                     ▷ Cross-chain request to $id'$
                run INIT($id'$)
                owner$^{id'} \leftarrow pk'$                              ▷ Activate authentication key
                received$^{id'} \leftarrow$ received$^{id'}$ :: $C$                  ▷ Update receiver's log
        case Execute($o$):
            try let $r$ = EXECUTEOPERATION($id, o$)                   ▷ Try to update state$^{id}$
            if $r = (m, id')$ then                  ▷ Was a cross-chain message requested by $o$?
                do asynchronously                                 ▷ Cross-chain request to $id'$
                    run INIT($id'$)
                    inbox$^{id'} \leftarrow$ inbox$^{id'} + m$              ▷ Add the message to the inbox of $id'$
                    received$^{id'} \leftarrow$ received$^{id'}$ :: $C$              ▷ Update receiver's log
        case Receive($m$):
            try inbox$^{id} \leftarrow$ inbox$^{id} - m$          ▷ Try removing the message from the local inbox
            try run $r$ = EXECUTEMESSAGE(id, m)                       ▷ Try to update state$^{id}$
            if $r = (m', id')$ then         ▷ Was another cross-chain message requested by $m$?
                do asynchronously                                 ▷ Cross-chain request to $id'$
                    run INIT($id'$)
                    inbox$^{id'} \leftarrow$ inbox$^{id'} + m'$             ▷ Add the message to the inbox of $id'$
                    received$^{id'} \leftarrow$ received$^{id'}$ :: $C$              ▷ Update receiver's log
        case ChangeKey($pk'$):
            owner$^{id} \leftarrow pk'$                                  ▷ Update authentication key
        case CloseChain:
            owner$^{id} \leftarrow \perp$                                      ▷ Make the chain inactive

function EXECUTEBLOCK($id$, $\tilde{T}$, $C$)
    for $T \leftarrow \tilde{T}$ do
        run EXECUTETRANSACTION($id$, $T$, $C$)

function BLOCKISVALID($id$, $\tilde{T}$)
    save state$^{id}$
    for $T \leftarrow \tilde{T}$ do
        try run EXECUTETRANSACTION($id$, $T$, $\perp$) while skipping the asynchronous blocks
    ensure inbox$_{-}^{id}$ is empty
    restore state$^{id}$
    return true
```

Algorithm 2 Validator service (single-owner chain)

```
 1: function HandleRequest($\mathsf{auth}[B]$)
 2:     let $\mathsf{Block}(id, n, h, \tilde{O}) = B$
 3:     ensure $\mathsf{owner}^{id} \neq \bot$                      ▷ The chain must be active
 4:     verify that $\mathsf{auth}[B]$ is valid for $\mathsf{owner}^{id}$        ▷ Check authentication
 5:     if $\mathsf{pending}^{id} \neq B$ then
 6:         ensure $\mathsf{pending}^{id} = \bot$
 7:         ensure $\mathsf{next\text{-}height}^{id} = n$
 8:         ensure $\mathsf{block\text{-}hash}^{id} = h$
 9:         ensure BlockIsValid($id$, $\tilde{O}$)
10:         $\mathsf{pending}^{id} \leftarrow B$                            ▷ Lock the chain on $B$
11:     return Vote($B$)                 ▷ Success: return a signature of the request

12: function HandleCertificate($C$)
13:     verify that $C = \mathsf{cert}[B]$ is valid
14:     let $\mathsf{Block}(id, n, h, \tilde{O}) = B$
15:     ensure $\mathsf{owner}^{id} \neq \bot$                  ▷ Make sure the chain is active
16:     if $\mathsf{next\text{-}height}^{id} = n$ then
17:         run ExecuteBlock($id$, $\tilde{O}$, $C$)
18:         $\mathsf{next\text{-}height}^{id} \leftarrow n + 1$                      ▷ Update block height
19:         $\mathsf{block\text{-}hash}^{id} \leftarrow \mathsf{hash}(B)$
20:         $\mathsf{pending}^{id} \leftarrow \bot$                  ▷ Make the chain available again
```

A Cross-chain communication

We now make precise the data structures used for inboxes as well as the practical implementation of cross-chain requests.

A.1 Messages and inboxes

We have seen that cross-chain messages m are executed after a command $\mathsf{Receive}(m)$ is included in a block of a receiving chain (Algorithm 1).[3]

Cross-chain messages. An *incoming message* (or simply *message*) is a pair $m = (id, \epsilon)$ including the following information:

- a chain id at the origin of the message;
- an *event* ϵ (defined next).

The event ϵ carried by m is a pair $\epsilon = (c, \mu)$ where

- $c = (n, i)$ is a *cursor* made of a blockheight n and an index i,
- μ is a message payload.

Here, the index i is understood as the index of the payload μ within the n-th block of the chain id. The payload μ is meant to be executed on the receiving chain by a specific application (also encoded in μ). Cursors are ordered lexicographically. If $c = (n, i)$, we write $c + 1 = (n, i + 1)$.

Inbox operations. For a particular node α and chain id', we write $\mathsf{inbox}^{id'}(\alpha)$ for the *inbox* made of messages received by a chain id' and waiting to be picked. Intuitively, an inbox can be thought as a set of messages m. Yet, in Section 2, we have seen that because validators may add and pick messages in different orders, convergence requires at least two sets of messages $\mathsf{inbox}^{id'}(\alpha) = I = (I_+, I_-)$.

In the practical implementation of Linera, we have chosen a more efficient approach that restricts the order in which messages can be received. Namely, messages must have increasing cursors $c = (n, i)$ for each origin id. Concretely, we see $\mathsf{inbox}^{id'}(\alpha)$ as a collection of partial inboxes $\mathsf{inbox}^{id',id}(\alpha)$. Each partial inbox $\iota = (S_+, S_-, c_+, c_-)$ is made of the following data:

- A set of added events S_+ (initially $\emptyset$),
- A set of removed events S_- (initially $\emptyset$),
- A minimum cursor to be added c_+ (initially $(0, 0)$),
- A mimimum cursor to be removed c_- (initially $(0, 0)$).

At all times, either S_+ or S_- must be empty. c_+ (resp. c_-) is meant to guarantee that added (resp. removed) events have increasing cursors.

[3]The current implementation of Linera assumes (without loss of generality) that the commands $\mathsf{Receive}(m)$ happen at the beginning of a block. This choice may be revisited in the future.

For any set of events S, we define $S\uparrow c = \{(c', e') \in S$ such that $c' > c\}$ and $S\uparrow_= c = \{(c', e') \in S$ such that $c' \geq c\}$. Let $\iota = (S_+, S_-, c_+, c_-)$ and $\epsilon = (c, \mu)$. Removing ϵ from ι is a (fallible) operation defined as

$$\iota - \epsilon = \begin{cases} (\emptyset, & S_- \cup \{\epsilon\}, \quad c_+, \quad c+1) \quad \text{if } c \geq c_- \text{ and } S_+\uparrow_= c = \emptyset \\ (S_+\uparrow c, & \emptyset, \quad c_+, \quad c+1) \quad \text{if } c \geq c_- \text{ and } \epsilon \in S_+ \\ \bot & \text{otherwise} \end{cases}$$

We note that the operation $S_+\uparrow c$ removes the event ϵ, as well as all events with a lower cursor. This allows a receiving chain to effectively skip incoming messages.

Adding ϵ to ι is a (fallible) operation defined as

$$\iota + \epsilon = \begin{cases} (S_+ \cup \{\epsilon\}, & \emptyset, \quad c+1, \quad c_-) \quad \text{if } c \geq c_+ \text{ and } S_- = \emptyset \\ (\emptyset, & S_-\uparrow c, \quad c+1, \quad c_-) \quad \text{if } c \geq c_+ \text{ and } (S_-\uparrow c) \cup \{\epsilon\} = S_- \\ \bot & \text{otherwise} \end{cases}$$

Importantly, the condition on S_- in the second case is stronger than $\epsilon \in S_-$ and implies that we only remove the element ϵ with the smallest cursor from S_-.

Finally, in relation to Algorithm 1, if $m = (id, \epsilon)$, removing m from $\mathsf{inbox}^{id'}(\alpha)$ (resp. adding m to it) consists in removing ϵ from the partial inbox $\mathsf{inbox}^{id',id}(\alpha)$ (resp. adding m to the partial inbox).

Outgoing messages and certificates. An *outgoing message* M is a tuple $M = (id', \mu)$ where

- id' is a target chain,
- μ is a message payload as before.

In practice, block certificates C are extended with the vector of outgoing messages $\tilde{M}$ scheduled by the execution of the block, namely: $C = \mathsf{cert}[B, \tilde{M}]$. This is useful for the implementation of cross-chain messages as well as the distributed auditing of chains.

A.2 Cross-chain requests and outboxes

Defining inboxes so as to ensure ordered delivery of messages is meant to simplify the implementation of an exactly-once delivery of cross-chain requests represented by the "do asynchronously { .. }" blocks in Algorithm 1.

Outboxes. Specifically, when a message $m = (id, \epsilon)$ is scheduled to be sent from id to a chain id', we mean to first add it to a local structure called an *outbox* and written $\mathsf{outbox}^{id',id}(\alpha)$.

Because messages are created by certified blocks that are executed in order, the current implementation of Linera defines an outbox simply as a set of block heights Q (initially $\emptyset$).

The *content* of an outbox $\mathsf{outbox}^{id',id}(\alpha) = Q$ is defined as the set of events $\epsilon = ((n, i), \mu)$ such that there exists a certified block $\mathsf{cert}[B, \tilde{M}]$ with height $n \in Q$ in id such that the outgoing message in position i of $\tilde{M}$ is $M = (id', \mu)$, and such that μ has not been confirmed to be received yet. We say that the event ϵ is contained in C with *target* id'.

When a new certified block B is executed at height n and creates its i-th outgoing message $M = (id', \mu)$, the event $\epsilon = ((n, i), \mu)$ must be added to the suitable outbox of id with target id'. Concretely, this is done by adding n to the set $Q = \mathsf{outbox}^{id',id}(\alpha)$.

Let Q be an outbox. To mark all the messages of Q as received up to a block height $\hat{n}$, we define the operation $Q\uparrow\hat{n} = \{n \in Q$ such that $n > \hat{n}\}$.

Cross-chain handshake. The current implementation of Linera starts a round of *handshakes* with all the relevant chains id' at the end of every run of the handler HandleCertificate on a certificate for a chain id (see Algorithm 2). Specifically, for every outbox $\mathsf{outbox}^{id',id}(\alpha) = Q$ such that $Q \neq \emptyset$, the following remote procedure calls (RPCs) are exchanged and processed by a dedicated handler HandleCrossChainRequest:

1. The worker of id sends to id' a RPC message $\mathsf{Update}(id', id, \tilde{C})$ containing a (non-empty) sequence of certificates $\tilde{C}$ issued by id and corresponding to the block heights $n \in Q$, by increasing order.

2. Upon receiving such a message $\mathsf{Update}(id', id, \tilde{C})$, the worker of id' extracts all the events ϵ with target id' contained in each certificate C in $\tilde{C}$. Then, it attempts to add each event to the inbox of id', by increasing order of cursor (i.e by increasing block heights and indices in a block). A message m may fail to be added if it was already added in a previous handshake; in this case, m is simply skipped.

3. The worker of id' replies to id with a RPC message $\mathsf{Confirm}(id', id, \hat{n})$ containing the highest processed block height so far $\hat{n}$ (if any).

4. Upon receiving $\mathsf{Confirm}(id', id, \hat{n})$, the worker of id marks the messages as received in the corresponding outbox: $\mathsf{outbox}^{id',id}(\alpha) \leftarrow \mathsf{outbox}^{id',id}(\alpha) \uparrow \hat{n}$.

In Step 3, $\hat{n}$ is obtained from the cursor c_+ of $\mathsf{inbox}^{id',id}(\alpha)$ as follows: $\hat{n} = \perp$ if $c_+ = (0, 0)$. If $c_+ = (n, 0)$ for some $n > 0$, then $\hat{n} = n - 1$. Otherwise, we have that $c_+ = (n, i+1)$ implies $\hat{n} = n$.

This translates into the pseudo-code of Algorithm 3.

Cross-chain requests without a message. In Algorithm 1, the operation OpenChain triggers a cross-chain request with a specific one-time effect. In practice, we introduce special messages to carry all requests using the same handshake protocol. In the example of OpenChain, the receiving chain is created during the cross-chain handshake when the message is about to be added to the inbox.

Algorithm 3 Internal cross-chain handshake protocol

```
1: function SendCrossChainRequests($id$)    ▷ Send all pending cross-chain requests from $id$
2:     for $Q = \mathsf{outbox}^{id',id}$ in $\mathsf{outbox}^{id}$ do    ▷ For every non-empty outbox of $id$
3:         if $Q \neq \emptyset$ then
4:             let $\tilde{C} = [$GetCertificate$(id, n)$ for $n$ in $Q]$    ▷ List certificates by increasing height
5:             send $\mathsf{Update}(id', id, \tilde{C})$    ▷ Send an update to $id'$

6: upon message $\mathsf{Update}(id', id, \tilde{C})$:    ▷ $id'$ received an update from $id$
7:     for $C$ in $\tilde{C}$ do
8:         for $\epsilon$ in GetEvents$(id', C)$ do    ▷ Obtain the events meant for us
9:             run Init$(id')$
10:            CheckImmediateEffect$(id', \epsilon)$    ▷ Apply any immediate effect
11:            try $\mathsf{inbox}^{id',id} \leftarrow \mathsf{inbox}^{id',id} + \epsilon$    ▷ Update the inbox
12:        $\mathsf{received}^{id'} \leftarrow \mathsf{received}^{id'} :: C$    ▷ Track the dependency
13:    let $\hat{n} =$ LatestAddedHeight$(\mathsf{inbox}^{id',id})$    ▷ Find the latest height from $(id)$
14:    if $\hat{n} \neq \bot$ then
15:        send $\mathsf{Confirm}(id', id, \hat{n})$    ▷ Reply with a confirmation

16: upon message $\mathsf{Confirm}(id', id, \hat{n})$:    ▷ $id$ received a confirmation from $id'$
17:    $\mathsf{outbox}^{id',id} \leftarrow \mathsf{outbox}^{id',id} \uparrow \hat{n}$    ▷ Discard all confirmed messages

18: function GetEvents$(id', C)$    ▷ Find the events contained in $C$ with target $(id', a)$
19:    $r \leftarrow []$
20:    let $C = \mathsf{cert}[B, \tilde{M}]$ and $B = \mathsf{Block}(id, n, h, \tilde{O})$
21:    for $0 \leq i < len(\tilde{M})$ do
22:        if let $(id', \mu) = M[i]$ then    ▷ If the $i$-th outgoing message matches
23:            let $\epsilon = ((n, i), \mu)$
24:            $r \leftarrow r :: \epsilon$    ▷ add an event
25:    return $r$

26: function CheckImmediateEffect$(id', \epsilon)$    ▷ Apply any immediate effect of the event
27:    let $(c, \mu) = \epsilon$
28:    if let $\mathsf{OpenChain}(id', pk') = \mu$ then
29:        $\mathsf{owner}^{id'} \leftarrow pk'$    ▷ Create the chain $id'$ immediately upon receiving the request

30: function LatestAddedHeight$(\iota)$    ▷ Find the latest height added to a partial inbox
31:    let $(S_+, S_-, c_+, c_-) = \iota$
32:    let $(n, i) = c_+$
33:    if $i > 0$ then
34:        return $n$
35:    else if $n > 0$ then
36:        return $n - 1$
37:    else
38:        return $\bot$
```

** Release notes **

Aug 16, 2023 – Version 2

- Clarify the definitions of public chains and how blocks may be produced in such chains.
- Make visible in the pseudo-code that messages may produce other messages.
- Swap the definitions of transactions and operations to be closer to the reference implementation [2] and the developer manual [1].
- Add an appendix to document the cross-chain communication protocols.
- Add section on user authentication.

Dec 19, 2022 – Version 1

- Initial release