1DLT: Rapid Deployment of Secure and Efficient EVM-Based Blockchains

Simone Bottoni ✉
RTM, Lugano, Switzerland

Anwitaman Datta ✉
Nanyang Technological University, Singapore, Singapore

Federico Franzoni ✉
Unaffiliated Researcher, Barcelona, Spain

Emanuele Ragnoli ✉
RTM, Lugano, Switzerland

Roberto Ripamonti ✉
RTM, Lugano, Switzerland

Christian Rondanini ✉
RTM, Lugano, Switzerland

Gokhan Sagirlar ✉
RTM, Lugano, Switzerland

Alberto Trombetta ✉
Insubria University, Varese, Italy

Abstract

Limited scalability and transaction costs are some of the critical issues that hamper a wider adoption of distributed ledger technologies (DLTs). That is particularly true for the Ethereum [58] blockchain, which, so far, has been the ecosystem with the highest adoption rate. Several solutions have been attempted in the last few years, most of which adopt the approach to offload transactions from the blockchain mainnet, a.k.a. *Level 1* (L1), to a separate network. Such solutions are collectively known as *Level 2* (L2) systems. While improving scalability, the adoption of L2 introduces additional drawbacks: users have to trust that the L2 system has correctly performed transactions or, conversely, high computational power is required to prove transactions' correctness. In addition, significant technical knowledge is needed to set up and manage such an L2 system. To tackle such limitations, we propose 1DLT[1]: a novel system that enables rapid deployment of an Ethereum Virtual Machine based (EVM-based) blockchain that overcomes those drawbacks.

2012 ACM Subject Classification Computer systems organization → Distributed architectures

Keywords and phrases Blockchain, EVM, Layer Two, Scalability, Network Fees

Digital Object Identifier 10.4230/OASIcs.Tokenomics.2022.3

Funding Anwitaman Datta's research is supported by Ministry of Education (MOE) Singapore's grant MOE-T2EP20120-0003 (award number #020931-00001).

1 Introduction

The current high demand for Ethereum [58] leads to slow transaction throughput (15-30 transactions per second [15]), expensive gas prices, and poor user experience for the majority of dapps (decentralised apps), Web3 projects, and end users. This limits the potential use cases, like in DeFi (decentralised finance), where high fees and scalability drawbacks enable only entities with vast economic power to trade profitably.

A notable example of extremely high gas prices and network congestion occurred with the launch of a new NFT for the *Bored Ape Yacht Club* metaverse [52]: during the launch, the Ethereum blockchain crashed due to traders outbidding each other by paying higher gas fees to execute their transactions faster. Users spent up to 7,000$ (2.6 ethers) to mint a 5,846$ NFT land deed for the virtual world, which sometimes resulted nevertheless in a failed transaction. A user trying to send 100$ in crypto between two wallets would need to pay a fee of 1,700$. As of July 2022, the cost of the above-mentioned NFT floats around 3,000$.

[1] Work done while all the authors were at QPQ AG.

© Simone Bottoni, Anwitaman Datta, Federico Franzoni, Emanuele Ragnoli, Roberto Ripamonti, Christian Rondanini, Gokhan Sagirlar, and Alberto Trombetta;
licensed under Creative Commons License CC-BY 4.0

Therefore, scaling solutions become crucial to increase network capacity in terms of speed and throughput. However, improvements to scalability should not be at the expense of decentralisation or trustlessness. Traditionally, scalability solutions are based on off-chain systems, collectively known as "Layer 2" (L2). L2 solutions are implemented separately from the "Layer 1" (L1) Ethereum mainnet, and do not require changes to its protocol. In L2 solutions, transactions are submitted to nodes of the L2 system instead of directly to L1 nodes. Thus, L2 solutions handle transactions outside the Ethereum mainnet and take advantage of its architectural features to ensure decentralisation and security. Existing L2 systems show a wide array of trade-offs among critical aspects like throughput, energy consumption, security guarantees, scalability, gas fees, and loss of trustlessness.

In this work, we present *One DLT* (1DLT), a novel, modular system for the rapid deployment of EVM-based blockchains, that avoids the pitfalls of many of the existing L2 solutions. Section 2 reviews the trade-offs of current solutions; Sections 3, 4, and 5 describe our system; Section 6 describes the Consensus-as-a-Service mechanism at the core of 1DLT; Section 7 show the 1DLT Bridge architecture; Section 8 exhibits a set of preliminary experimental results; finally, Section 9 concludes the work and describes our next steps.

2 Layer 2 limitations

There are several solutions available in the L2 ecosystem [13] (e.g., Optimistic Rollups [33], ZK-rollups [51], State channels [43], Sidechains [39]), with many different advantages and limitations. Due to space limitations, we do not present the main solutions adopted and we refer to the comprehensive surveys [53] and [54]. In the following, we list the most fundamental limitations and correlate them to some of the solutions adopted by the Ethereum ecosystem:

- **Limited expressive power**: some solutions do not support EVMs (e.g., several ZK-rollups, Plasma [34], Validium [48]); others support application-specific computations and require specialised languages (e.g., StarkWare's Cairo [55]);
- **Reduced trustlessness**: some solutions use operators and validators that can influence transaction ordering, leading to potential abuses (e.g., Optimistic Rollups, Sidechains);
- **Liveness requirement**: some solutions need to periodically watch the network or delegate this task to someone else to ensure security (e.g., Plasma);
- **High computational power to compute proofs**: some solutions require high computational power to compute proofs, which can be too expensive for dapps with little on-chain activity (e.g., ZK-rollups, Validium);
- **Reduced decentralisation**: some solutions adopt centralised methods to mediate the implementation of weak security schemes (e.g., Sidechains);
- **Limited throughput**: some solutions claim to theoretically achieve high transactions per second (tps) but are practically limited in their implementations (e.g, StarkWare [56] theoretically achieves 2,000 tps, while in real-world deployments is limited to 650 tps);
- **Not L2**: some solutions cannot be technically considered as L2 since they use separate consensus mechanisms that are not secured by the respective L1 (e.g., Sidechains). As such, these solutions cannot inherit from the L1 its security guarantees (e.g., resilience against chain tampering for Ethereum);
- **Private channels**: some solutions implement private channels, which is not a viable solution for infrequent transactions (e.g., State channels);
- **Long on-chain wait times**: some solutions require long wait times for on-chain transactions due to potential fraud challenges (e.g., Optimistic Rollups, Validium);
- **Data availability**: some solutions generate proofs that require off-chain data to be always available (e.g., Validium).

While the list above is not exhaustive (indeed, the L2 landscape is so dynamic that novel solutions, prototypes, and products are introduced to the market frequently), it is indicative of how, while there clearly exist attempts at overcoming these limitations, there is no single solution that can fix them all. It is important to note that a consequence of some of the limitations is the generation of security risks. Indeed, chains with relatively small ecosystems that provide consensus can lead to fallacies of abuse and fraud. Attackers, or the node maintainers themselves, may tamper with blockchain data ordering or validation, which allows them to redirect funds, perform flash loans or double-spending attacks, etc.

Lastly, in most solutions, users willing to create a private or public Ethereum network must rely on Ethereum *clients* (also known as *implementations*)[2] like Geth [20] and Erigon [8]. This approach requires the user to have significant technical knowledge and resources to maintain the nodes, with all the related costs and requirements of technical know-how.

3 An overview of 1DLT

This work has been inspired by the user experience of Cloud Service Providers (CSP) and Web-based applications, guided by the principles of DLTs. Indeed, in Cloud services, users are directed via graphical interfaces and dashboards through all processes (setup, configuration, billing, management, etc.). Such interaction is performed without any need of deep knowledge of the underlying technologies. Similarly, our goal is to provide a system that:

- streamlines and simplifies the deployment of a (public/private) EVM-based blockchain, as customarily happens for web-based services and CSP dashboards, without discarding the programmability of the EVM;
- maintains security while improving scalability and lowering fees;
- removes the risks associated with the L2 governance and fraud or abuse detection.

This is achieved with a modular, multilayered, cloud-native architecture (see Section 4) that decouples the transaction layer from the consensus layer. Thus:

- 1DLT connects to different blockchains and leverages their consensus mechanisms. We refer to this as *Consensus-as-a-Service* (CaaS) (see Section 6 for further details);
- 1DLT removes the risks associated with the L2 governance and fraud detection. Indeed, all transactions processed by 1DLT are sent to a L1 public blockchain, which is used as a consensus resource, allowing to inherit its security guarantees;
- 1DLT does not suffer from long wait times used in L2 to detect and avoid frauds. Fraud detection can be performed by checking receipts and confirmation messages of the public blockchain used as the consensus provider and local transactions' meta-data sent by CaaS;
- 1DLT is EVM-based, supports smart contracts written in the Ethereum programming language, Solidity, without requiring the adoption of L2 specific languages, such as Cairo;
- 1DLT transaction throughput is limited by that of the public blockchain that it connects to via CaaS. Thus, 1DLT outperforms Ethereum by connecting to blockchains with higher throughput and better scalability. Its performance can be further improved by connecting to different blockchains over time based on their load. The modular architecture makes it ready to plug in new, faster blockchain networks as they come into being;
- 1DLT allows to significantly reduce the transaction fees required to perform operations like payments, smart contract deployments, and token swaps, thanks to CaaS.

In the following, we show an example of user experience with 1DLT.

[2] A client is an implementation of Ethereum that verifies all transactions in each block, keeping the network secure and the data accurate [11].

▶ **Example 3.1.** Due to high transaction fees and long confirmation times, Alice wishes to move her dapp performing *NFTs Auctions* from the Ethereum mainnet to 1DLT to reduce operating costs. However, she does not want to change her codebase. To do that, Alice deploys a small, private EVM-based blockchain with 1DLT.

(i) Alice registers with the authentication system[3] and receives a redemption code for 1DLT blockchain creation;
(ii) Alice specifies the blockchain name and description, and token name and symbol. Then, with the redemption code, she creates a 1DLT Ethereum node in the private blockchain, configuring parameters such as cloud provider, virtual machine, etc.;
(iii) after specifying blockchain and nodes' parameters, Alice waits a short period for the execution of the setup procedure to start the deployment of her dapp;
(iv) Alice is now ready to deploy her dapp using the same procedure she used in Ethereum, that is, by sending a deployment transaction through the Web3 API;
(v) finally, upon receiving the deployment confirmation within few seconds, she and her customers are ready to interact with the dapp.

4 Architecture of 1DLT

1DLT has a modular architecture, consisting of two main components: a private EVM-based node (called *1DLT Ethereum Node*) and Consensus-as-a-Service (CaaS), a module that connects to external DLTs. When no confusion arises, from now on we refer to the 1DLT Ethereum Node simply as *Ethereum Node*. CaaS allows the Ethereum Node to leverage an external DLT to achieve consensus on transactions. Therefore, the 1DLT architecture abstracts and decouples the transaction layer from the consensus layer.

DLT adopts a very simple trust model in which the CaaS module is trusted. Hence, there are no checks from the Ethereum Node about the correctness of the responses from the CaaS module. Such trust assumption can be overcome by adding a verification mechanism to the Ethereum Node. In this work we focus on showing that 1DLT provides a low-cost scaling approach allowing high transaction outputs and fast transaction finality.

An overview of the 1DLT components and their interactions is shown in Figure 1. The structure of the Ethereum Node is detailed in the next section.

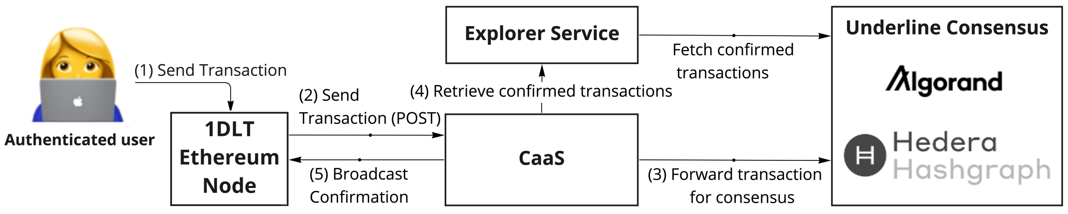

Authenticated user
(1) Send Transaction
1DLT Ethereum Node
(2) Send Transaction (POST)
(5) Broadcast Confirmation
Explorer Service
(4) Retrieve confirmed transactions
CaaS
Fetch confirmed transactions
(3) Forward transaction for consensus
Underline Consensus
Algorand
Hedera Hashgraph

Figure 1 1DLT architecture.

▶ **Example 4.1.** We continue with the scenario introduced in Example 3.1, where Alice sets up a node, and interacts with it by deploying a smart contract and sending a transaction. We expand on the operation flow sketched in Figure 1:

[3] A user must be authenticated to perform any action, thus a trusted authentication system is entitled to handle the user registration and management. Note that this trusted entity, while serving as a gateway for participants, is not relied upon for accountability of the actions of the participants. With our approach, the latter is achieved in a trustless manner using CaaS.

(i) Alice calls the Web3 API provided by the Ethereum Node to send a transaction, i.e. *eth_sendRawTransaction*;
(ii) the Ethereum Node receives the transaction and forwards it to CaaS using a POST message. This allows to achieve consensus on the transaction. The mechanisms used by CaaS to select a DLT for transaction dispatching is shown in Section 6;
(iii) CaaS forwards the transaction to the chosen DLT for confirmation;
(iv) when the transaction is confirmed, CaaS retrieves it using a blockchain explorer service (e.g., Hedera mirror service [24]);
(v) finally, CaaS sends the confirmation of the transaction to the Ethereum Node, which updates its state accordingly.

5 1DLT Ethereum Node

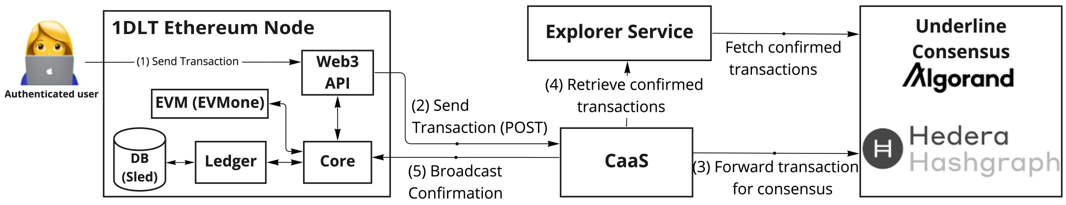


Figure 2 1DLT Ethereum node architecture.

While there are well-known and widely used implementations of Ethereum nodes, like Geth and Erigon, to overcome some of the limitations of Section 2, we engineered our own Ethereum node with a simpler architecture (see Figure 2). Section 5.2 describes the differences with a standard implementation. Usually, an Ethereum node contains a Web3 API, a state handling mechanism (i.e., the set of tries storing information on the state [29]), a database, an Ethereum Virtual Machine (EVM) [18], a p2p network, a transaction pool, and a consensus protocol. In the following, we describe the modules of our proposed architecture:

- **EVM module**: it is a sandboxed virtual stack machine that computes the system state transitions by executing an instruction specified in a transaction. In order to connect the node to a local, private EVM, the Ethereum Client-VM Connector API (EVMC) is used, as shown in Figure 3. The EVMC is the low-level interface between Ethereum Virtual Machines (EVMs) and Ethereum clients, which – on the EVM side – supports classic EVM1[4] and ewasm[5]. On the client side, EVMC defines the interface for accessing the Ethereum environment and state. A very relevant feature of EVMC is that nodes can connect with other non-Solidity based virtual machines.
  1DLT deploys a standalone C++ EVM implementation, called *EVMone* [19]. The EVMone EVM can be imported as a module by an Ethereum client and provides efficient execution of smart contracts written in an EVM-compliant language.
- **Web3 API module**: It handles incoming transactions and the communication with CaaS. It mostly works in the same way as in an Ethereum implementation, except for a customization that allows it to interact with CaaS. The module exposes a Web3-compatible API supporting modern Ethereum development tools and wallets (e.g., Metamask [30],

[4] Ethereum 1.x is a codename for a comprehensive set of upgrades to the Ethereum mainnet intended for near-term adoption.
[5] Ethereum flavoured WebAssembly is a subset of the WebAssembly format used for contracts in Ethereum.

Hardhat [22] and Web3.js [49]). E.g., the *eth_sendRawTransaction* API method is supported as in other implementations, but mining-related methods like *eth_isMining* are not supported, since consensus is handled through CaaS without any need for mining.

- **Ledger and state transitions module**: To store transactions and state, we use an Ethereum-compatible ledger implementation based on Merkle Patricia Trees (also known as (Merkle) Tries). This allows us to rely on the same structures as a standard Ethereum node (i.e., a State Trie, Receipt Trie, Transaction Trie, and Storage Trie [58]).
  Instead of the LevelDB [28] used in other Ethereum implementations, we opted for Sled [40], an embedded key-value store written in Rust optimised for modern hardware. It uses lock-free data structures to improve scalability and organises storage on disk in a log-structured way optimised for SSDs. We do not perform complex operations to achieve state transitions, such as the staged sync [9] in Erigon, as well as for block cutting. Since, CaaS validates transactions using an external consensus resource, it is possible to perform the block cutting in multiple ways. We opted to cut the block every Δ seconds (e.g., 10 seconds), after checking that the block is not empty.
- **Core module**: This module manages and coordinates the interaction of the other modules. It retrieves consensus updates of the processed transactions from CaaS (see Section 6) to perform state changes.

5.1 The execution flow

In what follows, we briefly describe the steps needed to perform a state update in a 1DLT Ethereum node upon receiving a transaction (see the sequence diagram in Figure 4).

1. the transaction is sent to the Ethereum node through the Web3 API;
2. the Web3 API module handles the transaction and sends a POST request to CaaS with the transaction wrapped inside the data field;
3. CaaS handles the transaction and connects to one of the chosen DLTs (e.g., Hedera) to confirm the transaction;
4. CaaS communicates with an external service (e.g. a Hedera mirror node) to retrieve the transaction confirmation;
5. CaaS then sends the confirmation message (i.e., the hash of the transaction with the consensus proof) of the transaction only to the Ethereum nodes that are part of the source network of the message.
6. the EVM of the Ethereum node executes the transaction, updating the state;
7. a block is then created if Δ seconds are passed (the time interval is a configurable parameter whose default is set to 10 seconds);
8. finally, the state transition result is stored in the Sled database.

5.2 Differences with current Ethereum implementations

Several relevant changes differentiate our node implementation from the standard one:

- **External Consensus**: in general, a consensus protocol has to be included in the internal architecture of a standard Ethereum node – like PoS for a mainnet, or PoA for a testnet node. Instead, we do not rely on an internal module, and delegate consensus to CaaS. This enables us to have the same liveness and safety guarantees of the chosen DLT while decreasing the complexity of the node.
- **No transaction pools**: in general, in a standard Ethereum node the transactions waiting for confirmation are placed into a transaction pool. Since our solution relies on an external consensus, all newly arrived transactions are forwarded directly to CaaS for confirmation without putting them in a queue. The benefits from this choice are a significantly simplified design and overall increased performance, as shown in Section 8.

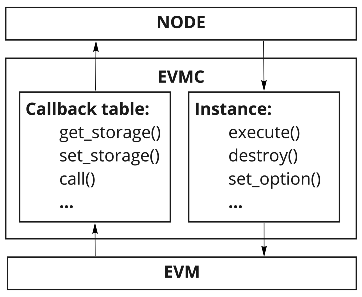


Figure 3 EVMC API.

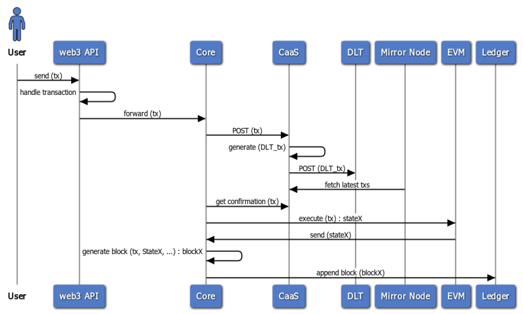


Figure 4 Sequence Diagram of a state update in a 1DLT Ethereum node.

- **Lightweight Core module**: as already mentioned in Section 5, the Core module is thoroughly simplified, since there is no transaction pool to manage, and it does not have complex state transitions (e.g., staged sync) as the consensus is retrieved from CaaS.
- **Difference in the Web3 support**: several methods are not supported by our implementation, such as those related to mining (e.g., $eth\_getMining$, $eth\_coinbase$), to uncles (e.g., $eth\_submitHashrate$), and to Ethereum protocol (e.g., $eth\_protocolVersion$).

6 Consensus-as-a-Service

Consensus-as-a-Service (CaaS) is the module that allows a 1DLT Ethereum node to access an external, public consensus protocol with an on-demand approach. Its key feature is the introduction of an abstract layer that enables the access to different DLTs through a single, uniform interface. This layer allows 1DLT Ethereum node to offload consensus complexity allowing it to achieve higher throughput and faster transaction finality with low transaction costs. Moreover, relying on an external consensus provider enables 1DLT to inherit the security model of the chosen DLT. Lastly, we remark that the CaaS approach eliminates any issue that may arise from the presence of a trusted third party, since transactions are public and easily auditable.

While in principle CaaS could attempt to tamper with handled transactions, such an attempt would make the transaction proof invalid, preventing the transaction execution, as each transaction proof required for an audit process can be retrieved from the target DLT. We acknowledge that this does not prevent CaaS to act in a malicious way and we leave as a future work the addition of a more efficient verification mechanism.

After choosing a suitable DLT, CaaS interacts with it by creating a channel that is used to publish messages containing transactions' information using the CaaS's DLT interface. The exchanged messages are stored in a time-series database (the current implementation uses timescale v2.6.0-pg1) to guarantee benefits over traditional relational database management systems like time-oriented features, higher data ingest rates and query performance. Additionally, each delivered message comes with the receipt of the transaction from the chosen DLT (e.g., an Hedera transaction receipt), which is an auditable proof that the transaction has been correctly processed by the DLT.

CaaS manages communication channels, I/O operations, and DB operations in a concurrent way, by spawning and managing multiple threads dedicated to the message exchange of different 1DLT networks. In particular, each 1DLT network has at least one dedicated communication channel, allowing high message processing throughput with low latency while also isolating multiple co-existing 1DLT networks from each other.

7 Bridge

Blockchains are siloed environments that cannot communicate with each other, as each network has its own protocols, native assets, data, and consensus mechanisms. Blockchain bridges, or cross-chain bridges [26][2], are a possible solution for enabling interoperability between different blockchains. The interoperability trilemma [45] allows for different bridge designs, for which, a non-standard classification can be based on [3]:

- **Trust model – How they work**: the type of authority used to synchronise the operations. The bridge is referred to as a "trusted bridge" if there is a central-trusted authority (e.g., Binance bridge [1]). If not, smart contracts make the bridge a "trustless bridge" by removing the necessity for a reliable third party (e.g., Connext [5], Hop [25], and other bridges with an atomic swap mechanism).
- **Validation – Validator or oracle-based bridges**: the type of mechanism the bridge relies on to validate cross-chain transfers, such as external validator or oracles.
- **Level – What they connect to**: the type of systems it connects to, such as a connection between blockchains or between a blockchain and an $L2$ system.
- **Sync – How they move assets**: the type of mechanism used to transfer assets between blockchains, such as *Lock and mint*, *Burn and mint*, or *Atomic swaps*.
- **Functionality – Their function**: the specialized interoperability task they are meant for, such as Chain-To-Chain, Multi-Chain, Specialized, Wrapped Asset, Data Specific, dapps Specific, and Sidechain.

1DLT offers a unique solution for bridges, enabling users to deploy a bridge by providing them with all the necessary tools and components (e.g., smart contracts). Since it operates via smart contracts, which serve as trusted parties, the 1DLT bridge belongs to the set of trustless bridges. It allows for the bi-directional transfer of ERC20 and ERC721 tokens between 1DLT nodes and EVM-compatible blockchains. 1DLT uses a *Lock and mint* mechanism: on the origin chain (e.g., Ethereum), a lock over the asset is performed, while on the destination chain (e.g., 1DLT), a mint is performed. Figure 5 provides a high-level breakdown of the bridge's core elements and how they interact. Essentially, the bridge is composed of a two of smart contracts, *Bridge.sol* and *Token.sol*, that are deployed on both the source and destination blockchain. Their interaction is coordinated via a cross-chain message dispatcher, called *Mediator*, using HTTP and WebSocket. The *Bridge* smart contracts implementation differ, as on the destination chain (i.e., *Bridge1DLT* for 1DLT) the contract design is for burn and mint, while on the origin chain is lock and withdraw (i.e., *BridgeETH*) for Ethereum). For the *Token* smart contract the implementation is the same for both the chains.

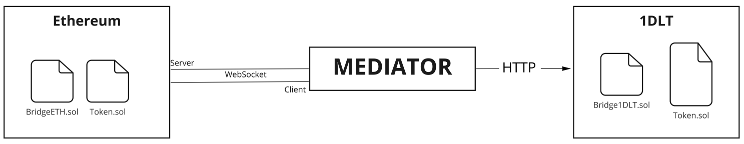


Figure 5 1DLT bridge architecture.

In what follows, we briefly describe the steps needed to perform the deposit of some ERC20 tokens from Ethereum to 1DLT (see Figure 6). The process relies on locking the asset on the source blockchain, and then mint the corresponding amount in the destination blockchain[6].

[6] To prevent the user from minting an arbitrary amount of token, only the bridge smart contract is entitled to call the mint method in the token smart contract.

1. The user sends a transaction to Ethereum, calling the *lock* method defined in the *BridgeEth* smart contract;
2. The transaction locks the tokens on Ethereum, transferring them to the *BridgeEth* address;
3. The *BridgeEth* emits a custom *Deposit* event with the address of the receiver on 1DLT and the amount;
4. The *Mediator* detects the event and retrieves the information;
5. The *Mediator* builds a transaction to call the *mint* method defined in *Bridge1DLT* with the event information as parameters;
6. The *Mediator* sends the new transaction to 1DLT;
7. 1DLT executes the transaction, which calls the *mint* method of *Bridge1DLT*;
8. The method calls the *mint* defined in *Token*;

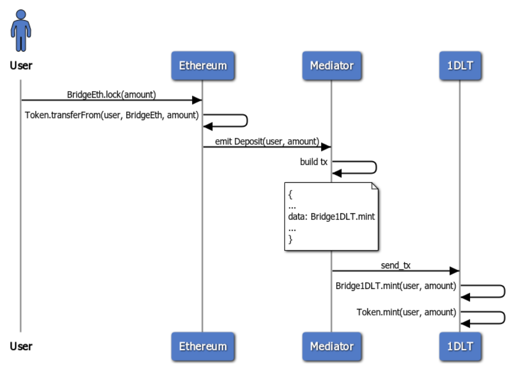


Figure 6 Deposit of ERC20 tokens from Ethereum to 1DLT.

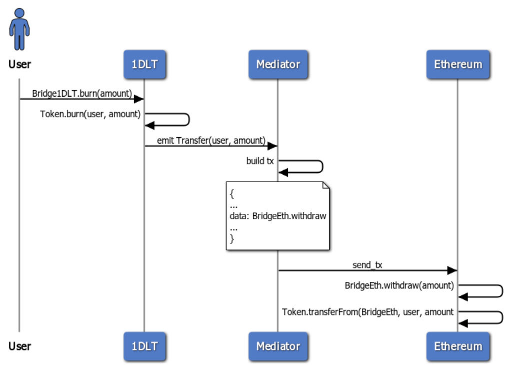


Figure 7 Withdraw of ERC20 tokens from 1DLT to Ethereum.

In what follows, we briefly describe the steps needed to perform the withdrawal of some ERC20 tokens from 1DLT to Ethereum (see Figure 7).

1. The user sends a transaction to 1DLT calling the *burn* method defined in *Bridge1DLT*;
2. The transaction burns the tokens on 1DLT;
3. The *Bridge1DLT* emits a custom event with the address of the receiver and the amount;
4. The *Mediator* detects the event and retrieves the information;
5. The *Mediator* builds a transaction to call the *withdraw* method defined in *BridgeEth* with the event information as parameters;
6. The *Mediator* sends the transaction to Ethereum;
7. Ethereum executes the transaction, which calls the *withdraw* method of *BridgeEth*;
8. The method calls the *transferFrom* defined in *Token*;

As virtually all complex, interacting systems, bridges are exposed to security risks related to:

- **Smart Contracts**: bugs in their code can be exploited for malicious behaviours;
- **Underlying Blockchain**: the underlying blockchain may be breached or act improperly;
- **Users**: users not following best practices can incur non-secure behaviours;
- **Censorship and Custodial**: bridge operators may act in malicious ways (e.g. they can suspend their activities or collude to gain sensitive information about the bridge's users)[7].

There are numerous examples of bridge attacks that resulted in multimillion dollar losses. [27]. It is worth to mention that bridge hacks mainly happen due to a vulnerability identified and exploited within the bridge contract, such as in the Wormhole attack [50] or the Optimism

[7] Applies to bridges that require the presence of trusted operators.

smart contract bug [32]. In the remaining cases, user mistakes take place, such as in the Optimism Wintermute case [31], where a Wintermute user inserted the wrong destination address for a transaction.

To overcome some of the previously mentioned risks, we give the user the ability to set up its 1DLT bridge without 1DLT system taking control of the bridge or custody of the assets. 1DLT offers reliable smart contracts that adhere to community-tested security best practices, such as Optimism [46] or Polygon[35]). Thus, an internal and external auditing procedure of the smart contracts, bridge, and the node is conducted to ensure the users' safety. For the external audit process, we use well-known auditors, such as CertiK[4], Hacken[21], and Trail of Bits [47]. As part of the internal audit process, we use a smart contract bytecode verification similar to the one of Etherscan [17] and Sourcify [42]. To this end, to have the smart contracts verified, a user must share with 1DLT the transaction hash of the exploited smart contracts via a dedicated page.

8 Experiments and Performance Discussion

A set of preliminary experiments were performed to benchmark and test 1DLT. We run experiments according to the following metrics:

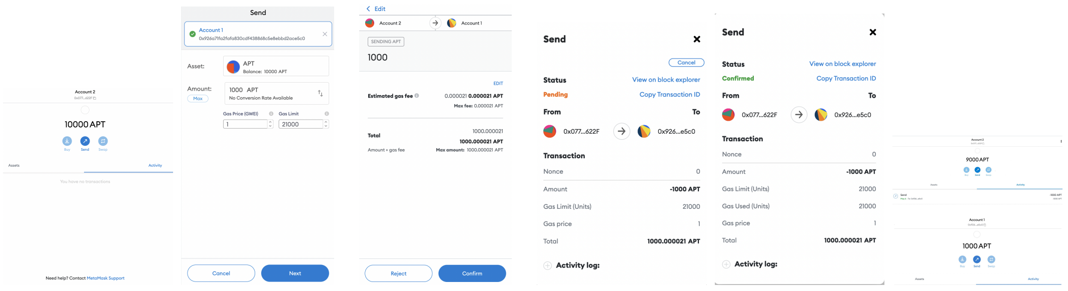
1. **Total transaction cost**: the cost of sending a transaction, which is computed as: $Cost_{total} = cost_{transaction} + fee_{gas} + fee_{DLT}$ where: $cost_{transaction}$ is the cost associated to the transaction, which may involve the execution cost of a smart contract or an amount of tokens; fee_{gas} and fee_{DLT} are the fee costs associated to the transaction from the node and for the target DLT. We note that the data field is where the difference between a transaction and an interaction lies; in a transaction, the field is empty; in an interaction, it has a value.
2. **Transaction finality**: the amount of time a user has to wait, on average, to obtain a confirmation of a transaction, measured in seconds. In the case of 1DLT, the finality time includes the transaction processing time and consensus finality time of the public DLT.
3. **Total smart contract deployment cost**: the cost of deploying a smart contract, which is computed as: $Cost_{total} = cost_{transaction} + fee_{gas} + fee_{DLT} + cost_{createContract} + cost_{data}$ where: $cost_{transaction}$ is the cost associated with the contract creation transaction; fee_{gas} and fee_{DLT} are the fee costs associated to the transaction from the node and for the target DLT; $cost_{createContract}$ is the cost associated to a contract creation, fixed to 32000 gas; and $cost_{data}$ is the cost associated to the contract complexity. As of today, we do not support only the legacy format (before EIP 2930 [7]).
4. **Throughput**: the transaction rate of a blockchain, measured in transactions per second (TPS). It is known that throughput is not the inverse of latency. For example, the transaction throughput for Bitcoin is about $7 tx/second$ [37] due to relatively small blocks and long block time. Instead, Ethereum has a short block time but tiny blocks, which results in a $15 tx/second$ [38]. 1DLT's throughput is limited by the total throughput of public blockchains that CaaS connects to. In fact, all transactions submitted to CaaS by the 1DLT Ethereum nodes are forwarded to the public blockchains like Hedera and Algorand. Therefore, 1DLT throughput varies proportionally with the throughput of the blockchain CaaS connects to.

The experiments are executed on an Azure Virtual Machine[8] configured as *Standard_D2_v3*[9], with 2 vCPUs, 8 GB of RAM, 256 GB SSD, and running *Ubuntu 21.10*. We use Hedera Consensus Service (HCS) [23] on the Testnet as the consensus resource. We simulate the Web3 API interaction using Web3.js API [49]. We use Metamask [30] as the wallet application to verify the state of the transactions and Hardhat as development environment [22], as it is the de-facto standard tool for developing dapps [41].

8.1 Total transaction cost

We consider the token in Alice's 1DLT network, with token name and symbol *Alice_Token* and APT, respectively. In Figure 8, we show the steps done from Metamask's user interface to transfer tokens from Alice to Bob: first, we specify Bob's account as the destination, the APT token as the asset, and 1,000 as the amount. Next, we check the calculated fees and send the transaction. Initially, the transaction state is on pending, then, after 6 seconds, the state changes from pending to confirmed, allowing the balance update for Alice and Bob.

The total cost ($Cost_{total}$) for Alice is as follows: fee_{Gas} is 0.000021 APT, $cost_{transaction}$ is 1000 APT, and fee_{DLT} is 0.00051779 *HBAR*, which is 0.00000003 APT (assuming that 1 ETH = 1 APT). So the cumulative cost is $1,000.00002103$ APT and the cumulative cost for the fees is 0.00002103 (0.056 USD). On the Ethereum Testnet (Ropsten [14]) the cumulative cost is 0.00005093 (total of 0.14 USD), while on the Ethereum mainnet, with a gas fee of 47 *Gwei*, it's 0.000819 (for a total of 2.95 USD).



Account 2
0x077...622F
10000 APT
You have no transactions
Need help? Contact MetaMask Support

Send
Account 1
0x926a71fa2fafa830cdf438868c5e8ebbd2ace5c0
Asset: APT Balance: 10000 APT
Amount: 1000 APT No Conversion Rate Available
Gas Price (GWEI) 1
Gas Limit 21000

Account 2 → Account 1
SENDING APT
1000
Estimated gas fee 0.000021 **0.000021 APT**
Max fee: 0.000021 APT
Total 1000.000021 **1000.000021 APT**
Amount + gas fee Max amount: 1000.000021 APT

Send
Status: Pending
From 0x077...622F To 0x926...e5c0
Transaction
Nonce 0
Amount -1000 APT
Gas Limit (Units) 21000
Gas price 1
Total 1000.000021 APT
Activity log:

Send
Status: Confirmed
From 0x077...622F To 0x926...e5c0
Transaction
Nonce 0
Amount -1000 APT
Gas Limit (Units) 21000
Gas Used (Units) 21000
Gas price 1
Total 1000.000021 APT
Activity log:

Account 2
9000 APT
Send -1000 APT

Account 1
1000 APT

Figure 8 Setup and send transaction from Alice to Bob state transition and account update.

8.2 Transaction finality

We evaluate the consensus finality of 1DLT using a client app that generates payment transactions, and submits them to a 1DLT Ethereum node in its 1DLT network. We compute the overall time from the generation of the transaction to the balance update in the Metamask wallet as: $Overall\_time = Generate\_Send_{tx} + 1DLT\_Finality_{tx} + Update\_wallet_{tx}$ where:

- $Generate\_Send_{tx}$ is the time our client application takes to generate and send the payment transaction to a 1DLT Ethereum Node of 1DLT;
- $1DLT\_Finality_{tx}$ is the time for 1DLT to process a transaction;
- $Update\_wallet_{tx}$ is the time the Metamask wallet takes to update the balance via a call sent by 1DLT.

[8] https://azure.microsoft.com/
[9] https://docs.microsoft.com/en-gb/azure/virtual-machine/dv3-dsv3-series

In Figure 9, we present the experiment results, showing the *Overall_Finality_Time* that we measured executing 200 transactions. We observe that average execution time for *Overall_Finality_Time* is *4.526* seconds.

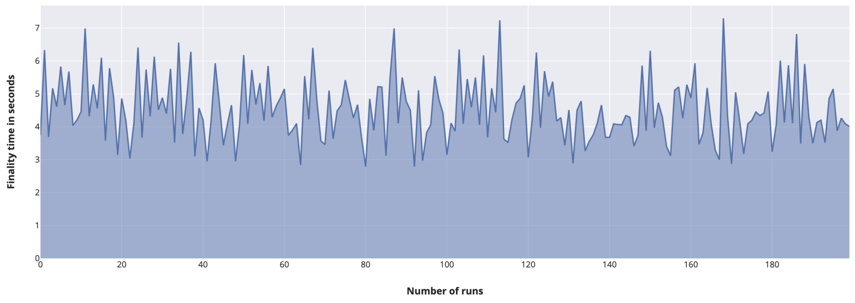


Figure 9 Transaction Finality experiment Diagram.

8.3 Smart contract deployment cost

We consider the example in 3.1 where Alice deploys the smart contract for her NFTs Auction dapp. To deploy the auction smart contract, we write a deployment script in JavaScript. We add the 1DLT node information, Node IP address, chain ID and private key of Alice account to the Hardhat configuration file, *hardhat.config.js*. Then, from terminal, we execute the deployment script with the command:

```
$ pnpm hardhat run --network AliceNetwork deploySmartContract.js
```

Once the deployment is completed, we receive the address of the created smart contract, like:

```
$ Contract deployed to address: 0x6cd7d44516a20882cEa2DE9f205bF401c0d23570
```

The transaction cost for deploying the smart contract on 1DLT is 0.000013402 APT (suppose that 1 ETH = 1 APT). On the Ethereum Testnet (Ropsten) the cumulative cost is 0.0015402 (for a total of 1.74 USD), while on the Ethereum mainnet, with a gas fee of 30 *Gwei*, it's 0.0117055 (for a total of 13.91 USD). Note that the cost to interact with a smart contract, that is, to call a method that changes the state (e.g., a set method), is calculated the same as a transaction since a setter method is implemented by sending a transaction.

8.4 Transaction per second (TPS)

We evaluate the performance of CaaS using Hedera as the consensus resource. We use a different Azure virtual machine configuration than before. We run CaaS on an Azure VM in the Switzerland North region configured as Standard DS3 v2, with 4 vCPUs, 14 GB of RAM, 1 TB SSD, and running Ubuntu 20.04. Then, to simulate the client, we use a VM configured with Standard D2s v3, 2 vCPUs, 8 GB memory, and running Ubuntu 20.04. In this experiment, we configured CaaS to run on a small VM to demonstrate that it is very lightweight and can achieve high performance even with this setup. Ideally, and in the production environment, CaaS will run on a Kubernetes cluster that allows to scale up with the increased transaction requests. The client runs a Python v3.8.10 script that generates a total of ten thousand transactions across five Hedera topics (which corresponds to five communication channels in CaaS), sends transactions to CaaS, and waits for confirmations from CaaS. Running the experiment 10 times with a single client results in an average of 1120 tps. The results are promising, as this experiment proves that a single client can process around 1120 transactions per second with a minimal CaaS setup as discussed above.

8.5 Discussion

The energy consumption of operating the Ethereum network has decreased by 99.9% after the Merge, occurred in mid September 2022 [44].

However, the gas fees' costs have not changed and this results in high costs for deploying Ethereum as a computing platform. In fact, each finalized block includes 30 million Gas, which is the amount of Gas used for all the transactions in a block [10]. The current transaction fees for 30 million of consumed gas is more than 1 Ether, which (as of July 2022) is valued at 1,479 USD. This implies that the computation costs of the Ethereum Network are around 133 USD per second, which is ~25 times more than 15 days of an EC2 instance (currently around 20 USD).

In order to estimate the cost of the Ethereum network itself (measured in gas), following [57], we consider a very basic computational task like adding two 256-bit integers. Since this operation costs 3 Gas [12] and the Ethereum network's total compute is 2 million gas/second, the Ethereum network may perform 600,000 additions per second. In contrast, Raspberry Pi 4 [36], a 45 USD single-board computer with four processors running at 1.5 GHz, can perform around 3,000,000,000 additions per second. As such, the Ethereum network, considered as a general-purpose computational environment, has roughly 1/5,000 of the computing power of a Raspberry Pi 4. At the current gas price, this means that performing 256-bit additions on the Ethereum network, costs about 60 USD per month.

Ethereum as a computational environment has the drawback to be expensive, in terms of gas fees. Since 1DLT follows a modular approach and separates the EVM-based computational layer from the consensus layer, it minimises energy consumption and computational effort. Thanks to the deployment of CaaS, the consensus engine does not require a mining algorithm in the consensus retrieval. Additionally, 1DLT offers the same level of computational power as Ethereum at a significantly lower cost, as shown in experiments 1 and 3 in Section 8. As an example, the cost to execute 50,000 transactions is 0.04 USD.

9 Conclusion and Future work

Scaling solutions are crucial to increase the Ethereum network's capacity in terms of speed and throughput, but they come at the cost of reduced decentralisation, increased transaction finality times. 1DLT, inspired by the user experience of Cloud Service Providers (CSP) and Web-based applications, overcomes the limitations of the existing scaling solutions enabling low gas fees, high transaction throughput, and fast transaction finality. Additionally, 1DLT removes the risk associated with the L2 governance and fraud detection, since all the transactions processed in 1DLT networks are submitted to the consensus protocols of L1 public DLTs. Lastly, the programmability and user experience of the Ethereum ecosystem is maintained thanks to an EVM-based architecture.

We demonstrated the feasibility of our architecture with a set of preliminary experiments in Section 8, benchmarking transaction costs and finality.

Future work includes: (i) a proper experimental benchmark to execute advanced experiments that fully accounts for the real-world landscape of L2 solutions and 1DLT; (ii) Extend the 1DLT Ethereum Node with a verification mechanism for proving the correctness of Caas module's responses; (iii) enhance the bridge with support for blockchains that are not compatible with EVM; (iv) the integration of 1DLT with Trusted Execution Environments; (v) full integration with a proprietary wallet that will enhance the Metamask solution; (vi) provision of user tools like Etherscan [16] for Ethereum or DragonGlass [6] for Hedera, to audit the status of the blockchains in the 1DLT ecosystem.

References

1 Binance bridge. https://www.bnbchain.org/en/bridge.
2 Blockchain bridges. https://ethereum.org/en/developers/docs/bridges.
3 Blockchain bridges classification. https://li.fi/knowledge-hub/bridge-classification.
4 CertiK. https://www.certik.com.
5 Connext bridge. https://bridge.connext.network.
6 DragonGlass. https://app.dragonglass.me.
7 EIPS 2930. https://eips.ethereum.org/EIPS/eip-2930.
8 Erigon. https://github.com/ledgerwatch/erigon.
9 Erigon stage sync. https://github.com/ledgerwatch/erigon/blob/devel/eth/stagedsync.
10 Ethereum gas. https://ethereum.org/en/developers/docs/gas.
11 Ethereum nodes. https://ethereum.org/en/developers/docs/nodes-and-clients.
12 Ethereum opcodes. https://ethereum.org/it/developers/docs/evm/opcodes.
13 Ethereum Scaling. https://ethereum.org/en/developers/docs/scaling.
14 Ethereum testnets. https://ethereum.org/en/developers/docs/networks.
15 Ethereum TPS. https://ethtps.info.
16 Etherscan. https://etherscan.io.
17 Etherscan smart contract verification. https://etherscan.io/verifyContract.
18 EVM. https://ethereum.org/en/developers/docs/evm.
19 EVMone. https://github.com/ethereum/evmone.
20 Geth. https://geth.ethereum.org/docs.
21 Hacken. https://hacken.io.
22 Hardhat. https://hardhat.org.
23 Hedera Consensus Service. hedera.com/consensus-service.
24 Hedera mirror service. https://hedera.com/learning/hedera-hashgraph/what-is-the-hedera-mirror-network.
25 Hop. https://app.hop.exchange.
26 Introduction to blockchain bridges. https://ethereum.org/en/bridges/.
27 Leaderboard of Ethereum bridge attacks. https://rekt.news/leaderboard.
28 Leveldb database. https://github.com/google/leveldb.
29 Merkle Patricia Trie. https://eth.wiki/fundamentals/patricia-tree.
30 Metamask. https://metamask.io.
31 Optimism Attack. https://cointelegraph.com/news/optimism-loses-20m-tokens-after-l1-and-l2-confusion-exploited.
32 Optimism Bounty. https://cryptoslate.com/critical-bug-in-ethereum-l2-optimism-2m-bounty-paid.
33 Optimistic Rollups. https://ethereum.org/en/developers/docs/scaling/optimistic-rollups.
34 Plasma. https://ethereum.org/en/developers/docs/scaling/plasma.
35 Polygon-Ethereum Bridge. https://docs.polygon.technology/docs/develop/ethereum-polygon/getting-started.
36 Raspberry. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf.
37 Real time Bitcoin finality. https://statoshi.info/d/000000006.
38 Real time Ethereum finality. https://ethtps.info.
39 Sidechains. https://ethereum.org/en/developers/docs/scaling/sidechains.
40 Sled database. https://github.com/spacejam/sled.
41 Solidity report. https://blog.soliditylang.org/2022/02/07/solidity-developer-survey-2021-results.
42 Sourcify. https://docs.sourcify.dev/docs/intro.
43 State Channels. https://ethereum.org/en/developers/docs/scaling/state-channels.
44 The Ethereum merge. https://ethereum.org/en/upgrades/merge.

45 The interoperability trilemma. `https://blog.connext.network/the-interoperability-trilemma-657c2cf69f17`.
46 The Optimism bridge. `https://ethereum.org/en/developers/tutorials/optimism-std-bridge-annotated-code`.
47 Trail of Bits. `https://www.trailofbits.com`.
48 Validium. `https://ethereum.org/en/developers/docs/scaling/validium`.
49 web3.js API. `https://web3js.readthedocs.io/en/v1.7.4/`.
50 Wormhole hack. `https://rekt.news/wormhole-rekt`.
51 Zk Rollups. `https://ethereum.org/en/developers/docs/scaling/zk-rollups`.
52 Bored ape crush Ethereum. `https://www.cnet.com/personal-finance/crypto/bored-ape-yacht-club-just-broke-the-ethereum-blockchain`, 2022.
53 Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In *Financial Cryptography and Data Security - 24th International Conference, FC*, Lecture Notes in Computer Science, pages 201–226. Springer, 2020.
54 Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies. *IACR Cryptol. ePrint Arch.*, page 352, 2019.
55 Starkware. Starkware Cairo. `https://starkware.co/cairo`.
56 Starkware. Starkware Libs. `https://github.com/starkware-libs`.
57 Nicholas Weaver. The Web3 Fraud. `https://www.usenix.org/publications/loginonline/web3-fraud`, 2021.
58 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.