QPQ 1DLT: A SYSTEM FOR THE RAPID DEPLOYMENT OF SECURE AND EFFICIENT EVM-BASED BLOCKCHAINS

Simone Bottoni, Anwitaman Datta, Federico Franzoni, Emanuele Ragnoli, Roberto Ripamonti, Christian Rondanini, Gokhan Sagirlar, and Alberto Trombetta

QPQ

eragnoli@qpq.io, crondanini@qpq.io, gsagirlar@qpq.io

V1.0 05-08-2022 Zug/London/Dublin

Abstract

Limited scalability and transaction costs are, among others, some of the critical issues that hamper a wider adoption of distributed ledger technologies (DLT). That is particularly true for the Ethereum [1] blockchain, which, so far, has been the ecosystem with the highest adoption rate. Quite a few solutions, especially on the Ethereum side of things, have been attempted in the last few years. Most of them adopt the approach to offload transactions from the blockchain mainnet, a.k.a. *Level 1* (L1), to a separate network. Such systems are collectively known as *Level 2* (L2) systems. While mitigating the scalability issue, 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: a novel system that enables rapid and trustless deployment of an Ethereum Virtual Machine based (EVM-based) blockchain that overcomes those drawbacks.

Keywords— Blockchain, EVM, Layer Two, Scalability, Network Fees

1 Introduction

The current high demand for Ethereum [1] leads to slow transaction throughput (15-30 transactions per second [2]), 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 and very recent example of extremely high gas prices and network congestion occurred with the launch of a new NFT for the *Bored Ape Yacht Club* metaverse [3]. Indeed, during the launch, the Ethereum blockchain crashed due to traders outbidding each other by paying higher gas fees in order to execute their transactions at higher speeds. Ethereum users would have spent up to 7,000$ (2.6 ethers) to mint a 5,846$ NFT land deed for the virtual world, which in some cases resulted nevertheless in a failed transaction. Likewise, a user trying to send 100$ in crypto between two wallets would need to pay a fee of 1,700$. As of July 7 2022, the cost of the above-mentioned NFT floats around 3,000$.

As it is well known, the Ethereum blockchain requires a massive amount of power for its operations. In fact, its annual energy consumption is estimated around 112 TWh [4], which is comparable to that of Austria. A single transaction over Ethereum is equivalent to the power consumption of an average US household over 9 days [5]. However, such huge amount of power does not result in a comparable computational power. In fact, almost all the computational power is used for computing hash values, with the power left for other operations being just a tiny fraction of the computational power of a Raspberry Pi 4 [6], see Section 8.1 for more details.

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 with the introduction of a trusted third party. 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 the existing Ethereum protocol. In L2 solutions, transactions are submitted to the nodes of the L2 system instead of directly to L1 nodes. Thus, L2 solutions handle transactions outside the Ethereum mainnet and take advantage of the architectural features of the mainnet to allow high decentralisation and security. The 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. The rest of this paper is organised as follows: 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 [7] (e.g., Optimistic Rollups [8], ZK-rollups [9], State channels [10], Sidechains [11]), with a lot of variation in terms of advantages and limitations. In the following, we list the most fundamental limitations, and we 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 [12], Validium [13]); other solutions support application-specific computations and require specialised languages (e.g., StarkWare's Cairo [14]);
- **L2 nodes**: some solutions use operators and validators that can influence transaction ordering, reversing the principle of trustlessness that might lead to abuses or frauds (e.g., Optimistic Rollups, Sidechains);
- **Liveness requirement**: some solutions need to periodically watch the network or delegate this responsibility 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 [15] theoretically achieves 2,000 tps, whereas the actual limit in real-world deployments is 650 tps);

- **Not L2**: some solutions cannot technically be 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 by no means exhaustive (indeed, the L2 landscape is so dynamic that novel solutions, prototypes and products are being introduced to the market frequently), it is indicative of how, while there clearly exist attempts at overcoming those limitations, there is not a 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 to take advantage, such as redirect funds, perform flash loans or double-spending attacks, etc.

Last but not least, in most of the solutions above, users aiming at creating a private or public Ethereum network must rely on Ethereum *clients* (also known as *implementations*)[1] like Geth [17] and Erigon [18]. This approach requires the user to have a 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, while CSP dashboards might not be perfect, a user is directed via graphical interfaces and dashboards through the process that enables to complete the setup, configuration, billing, and management of the chosen service. The interaction with such an interface is performed without any need of deep knowledge of the underlying technologies.

Hence, our goal is to provide a system that:

- has a very low technical barrier of entry 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 the security and trustlessness of DLTs while improving scalability and lowering gas 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 in order to leverage 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 the L1 public blockchain, which is used as a consensus resource. resource. This also allows to inherit the security guarantees of the L1 public blockchain that provides consensus;
- 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, and it directly supports smart contracts written in the Ethereum programming language, Solidity, and does not require the adoption of L2 specific languages, such as Cairo[2];
- 1DLT maintains trustlessness. Since every transaction is sent to a public blockchain via CaaS, 1DLT is as decentralised as the blockchain to which CaaS connects to;
- 1DLT transaction throughput is limited by the throughput of the public blockchain that it connects to via CaaS. Therefore, 1DLT outperforms Ethereum by connecting to blockchains with higher throughput and better scalability. Additionally, 1DLT performance can be improved by connecting to different blockchains over time based on load on a given network, and furthermore, the modular architecture makes it ready to leverage on new, faster blockchain networks as and when they come into being;

[1] A client is an implementation of Ethereum that verifies all transactions in each block, keeping the network secure and the data accurate [16]

[2] https://starkware.co/cairo/

- 1DLT allows to significantly reduce the transaction fees required to perform operations like payments, smart contract deployments, and token swaps, thanks to CaaS.

The following example is a snippet of the user experience with 1DLT.

Example 3.1. Due to the high transaction fees and long transaction confirmation times, Alice wishes to move her dapp performing *NFTs Auctions* from the Ethereum mainnet to another blockchain, that is 1DLT. However, she does not want to change her codebase. To achieve that, Alice deploys a small, private EVM-based blockchain with 1DLT, formed by a pair of nodes.

(i) Alice registers with the QPQ authentication system[3] and receives a redemption code for 1DLT blockchain creation;

(ii) Alice specifies the blockchain name, description, token name, and token symbol. Then, with the redeemed code, she creates the first QPQ Ethereum node of the private blockchain, choosing configuration parameters such as cloud provider, virtual machine, hostname, 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 Web3 API and process that she used in Ethereum, which in this case consists in sending a deployment transaction through the Web3 API;

(v) finally, upon receiving the deployment confirmation within instants, she and her customers are ready to interact with the dapp.

4 Architecture of 1DLT

The architecture of 1DLT is based on a modular approach, and it has two main components: a private EVM-based node (called *QPQ Ethereum Node*) and Consensus-as-a-Service (CaaS), a module that connects to external consensus providers. The CaaS module allows the QPQ Ethereum Node to access an external DLT and leverage its consensus mechanism. Therefore, the 1DLT architecture abstracts and decouples the transaction layer from the consensus layer.

A high-level overview of the components of 1DLT and their interactions is shown in Figure 1.

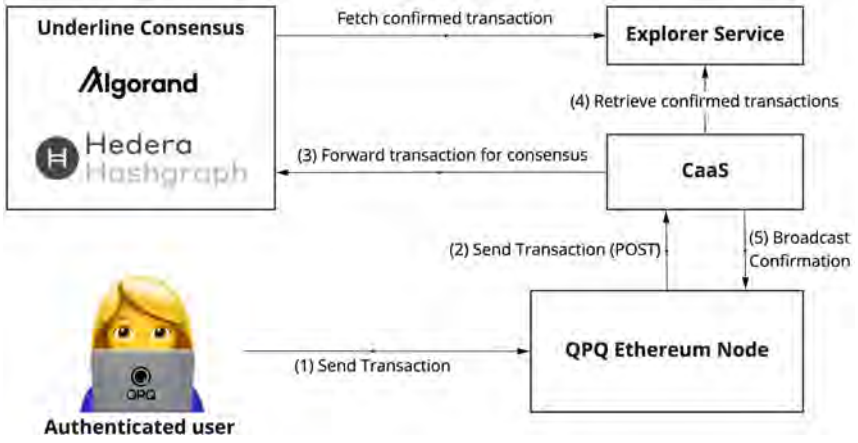

Figure 1: QPQ 1DLT architecture

Example 4.1. We continue with the scenario introduced in Example 3.1, in which Alice did the setup of 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:

(i) Alice calls the Web3 API provided by the QPQ Ethereum node to send a transaction, i.e. *eth_send_raw_transaction*;

(ii) the QPQ Ethereum node receives the submitted transaction and forwards it to the Consensus-as-a-Service (CaaS) handler using a POST message. The QPQ Ethereum node leverages CaaS to achieve consensus for the transaction. We reiterate that CaaS may operate with different consensus protocols, and we refer to Section 6 for the explanation of how CaaS chooses a DLT for transaction dispatching;

(iii) CaaS forwards the transaction to the chosen DLT to achieve consensus;

(iv) then, once the transaction is confirmed, CaaS retrieves it using a blockchain explorer service (e.g., Hedera mirror service [19]);

(v) finally, CaaS broadcasts the confirmation of the transaction to the QPQ Ethereum node, such that it can update its state.

[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.

5 QPQ Ethereum Node

While there are well-known and widely used implementations of Ethereum nodes, notably Geth and Erigon, in order to overcome some of the limitations of Section 2, we engineered our own Ethereum node with a much simpler architecture (see Figure 2). Section 5.2 describes the differences with a standard implementation. Generally speaking, an Ethereum node contains the following components: a Web3 API, a state handling mechanism (that is, the set of tries storing the information composing the state [20]), a database, an Ethereum Virtual Machine (EVM) [21], a p2p network, a transaction pool, and a consensus protocol. In the following, we present a detailed description of the modules of our proposed architecture:

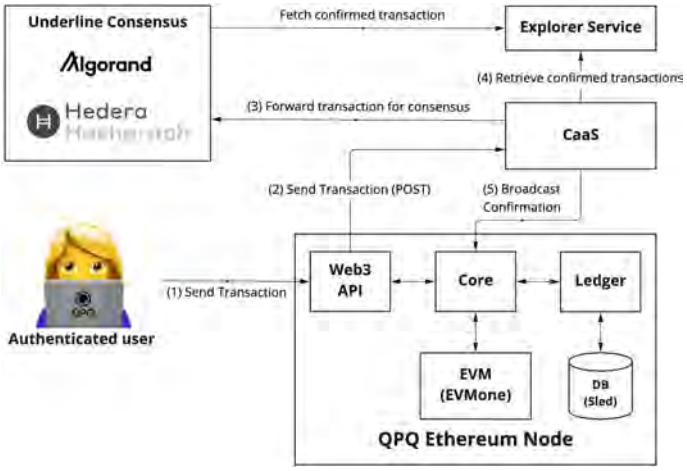

Figure 2: QPQ Ethereum node architecture

- **EVM module**: it is a sandboxed virtual stack machine. The purpose of the EVM module is to compute the new system state by executing an instruction specified in a smart contract. 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 lies in the fact that nodes can connect with other non-Solidity based virtual machines.

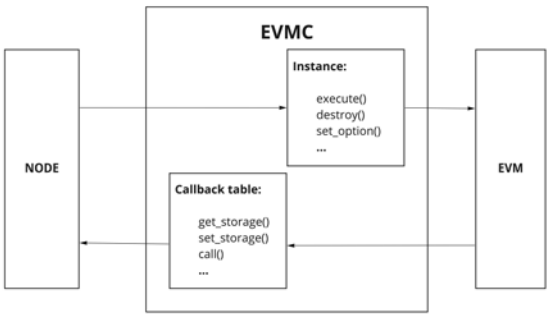

Figure 3: EVMC API

  1DLT deploys a standalone C++ EVM implementation, called *EVMone* [22]. 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 scripting language.

- **Web3 API module**: it handles incoming transactions and the communication with the CaaS handler. This module 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 [23], Hardhat [24] and Web3.js [25]). As an example, the API supports *eth_send_raw_transaction* the same way as other

[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 (wasm) format used for contracts in Ethereum.

Ethereum implementations do. On the other hand, it does not support mining-related methods like $eth\_isMining$, since we do not have nor need a consensus mechanism that involves mining.

- **Ledger and state transitions module**: regarding the ledger used for storing transactions' results, we opted for an Ethereum-compatible persistent ledger implementation using Merkle Patricia Trees (also known as (Merkle) Tries). This enables us to rely on the same storage structure of a standard Ethereum node (that is, a State Trie, Receipt Trie, Transaction Trie, and Storage Trie [1]).

  To this end, instead of the LevelDB DBMS [26] used in other Ethereum implementations, we opted for Sled [27], an embedded key-value store written in the Rust programming language. Sled provides atomic single-key operations, including compare and swap operations. It is designed to be used as a construction component to build larger stateful systems. Sled is 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 [28] in Erigon, as well as for block cutting. As previously discussed, CaaS validates the transactions using a public consensus resource, and this enables us to update the ledger state (EVM execution) and the block cutting directly without needing the consensus on the operation again. This enables us to perform the block cutting in multiple ways, without any constraints. To this end, we opted to cut the block every Δ seconds (e.g., 10 seconds), after checking that the block is not empty.
- **Core module**: The *Core* module manages and coordinates the interaction of the QPQ Ethereum Nodes' modules. It retrieves consensus updates of the processed transactions from the Consensus as a Service module (see section 6) to perform state changes. Unlike other Ethereum node implementations (e.g. Erigon) that perform complex operations for state updates (e.g. Erigon performs "stages sync"), a QPQ Ethereum Node can directly update the state after retrieving the consensus updates of transactions from CaaS.

5.1 The execution flow

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

1. The transaction is sent to the QPQ 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 blockchains (e.g., Hedera) to reach consensus on the transaction;
4. CaaS communicates with an external service (e.g. a Hedera mirror node) to retrieve the transaction confirmation;
5. CaaS then broadcasts the confirmation message (i.e., the hash of the transaction with the consensus proof) of the transaction to the QPQ Ethereum node;
6. the EVM of the 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.

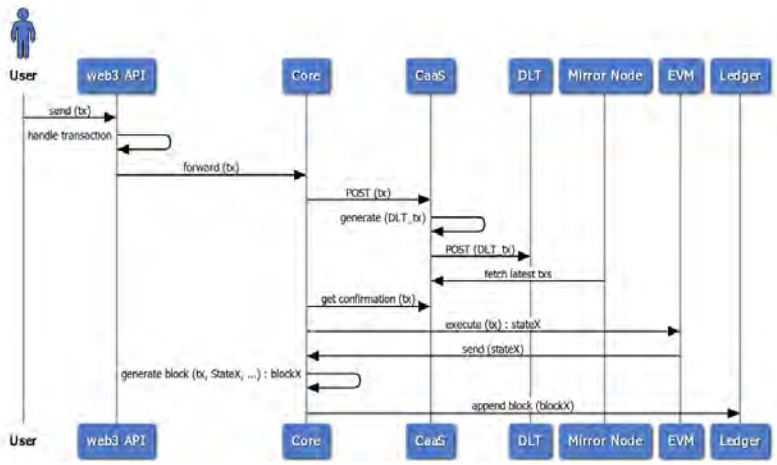

Figure 4: Sequence Diagram of a state update in a QPQ Ethereum node

5.2 Differences with current Ethereum implementations

Our approach demands that more than one protocol may be available for transaction consensus. As such, several relevant changes have been introduced that differentiate the node implementation used in 1DLT from the standard one:

- **External Consensus**: in general, a consensus protocol has to be included in the internal architecture of an Ethereum node – like PoW 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 level of liveness and safety guarantees of the chosen DLT while decreasing the overall complexity of the node.
- **No transaction pools**: in general, in an Ethereum node the transactions waiting for confirmation by means of an internal consensus mechanism are placed into a transaction pool. Since our solution relies on an external consensus via CaaS, 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.
- **Lightweight Core module**: as already mentioned in the previous section, 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**: having an external consensus provider implies that several methods are not supported by our implementation. In particular, methods related to mining (e.g., $eth\_getMining$, $eth\_submitHashrate$, $eth\_coinbase$), to uncles (e.g., $eth\_submitHashrate$), and to Ethereum protocol (e.g., $eth\_protocolVersion$) are not supported.

6 Consensus-as-a-Service

Consensus-as-a-Service (CaaS) is the module that allows a QPQ 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. Such a layer allows 1DLT networks' nodes to offload consensus complexity by enabling them to achieve better performance, such as 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. Last but not least, 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 CaaS could attempt to tamper with handled transactions, however, it would make the transaction proof invalid, preventing the transaction execution. In fact, CaaS cannot tamper with handled transactions in any way, as each transaction proof required for an audit process can be retrieved from the target DLT.

After having chosen a suitable DLT, CaaS interacts with it by creating a channel, that is used to publish messages that contain transactions' information using the CaaS's DLT interface. The exchanged messages are stored in a time-series database (in the current implementation, we use timescale v2.6.0-pg1) to guarantee benefits over traditional Relational database management systems (RDBMS) or vanilla PostgreSQL; these benefits include 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 itself.

CaaS manages communication channels, I/O operations, and DB operations concurrently. In essence, CaaS spawns and manages multiple threads dedicated to the message exchange of different 1DLT networks. In particular, each 1DLT network has at least one dedicated communication channel managed by CaaS, allowing high message processing throughput with low latency. Having dedicated communication channels also allows multiple 1DLT networks to co-exist while isolating them 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 [29][30], are a possible solution for enabling interoperability between different blockchains. The interoperability trilemma [31] allows for different bridge designs, for which, a non-standard classification can be based on [32]:

- **Trust model - How they work**: that is, 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 [33]). If not, smart contracts make the bridge a "trustless bridge" by doing away with the necessity for a reliable third party (e.g., Connext [34], Hop [35], and other bridges with a simple atomic swap mechanism).
- **Validation - Validator or oracle based bridges**: that is, the type of mechanism the bridge relies on to validate cross-chain transfers, such as external validator or oracles.
- **Level - What they connect to**: that is, the type of systems it connects to. If the connection is between blockchains or between a blockchain and an *L2* system.
- **Sync mechanism - How they move assets**: that is, the type of mechanism used to transfer assets between blockchains, such as *Lock and mint*, *Burn and mint*, or *Atomic swaps*.
- **Functionality - Their function**: that is, the (more or less) specialized interoperability task they are meant for, such as Chain-To-Chain Bridges, Multi-Chain Bridges, Specialized Bridges, Wrapped Asset Bridges, Data Specific Bridges, dapps Specific Bridges, and Sidechain Bridges.

1DLT offers a unique solution for bridges. We enable users to deploy the bridge by providing them with all the tools and components (e.g., smart contracts). Since the bridge is operated via smart contracts, which serve as a trusted party, 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 set of smart contracts, *Bridge.sol* and *Token.sol*, that are deployed on both the source and destination blockchain. Their interaction is coordinated via a bridge API, called *Mediator*, using HTTP and WebSocket. The *Bridge* smart contract the 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 *Token* 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 in the destination blockchain mint the amount[6].

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*;

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

[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.

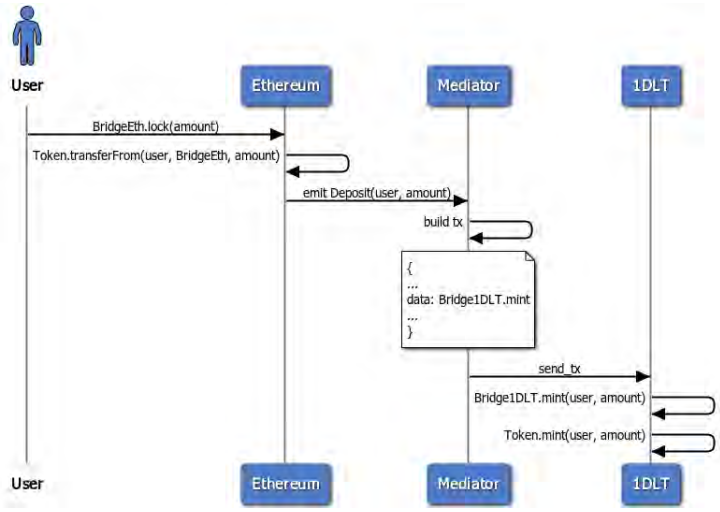

Figure 6: Deposit of ERC20 tokens from Ethereum to 1DLT

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*;

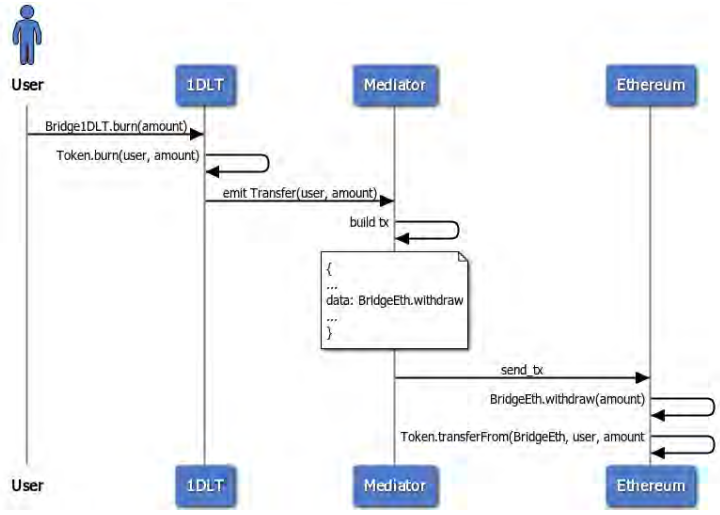

Figure 7: Withdraw of ERC20 tokens from 1DLT to Ethereum

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

- **Smart Contracts**: bugs in their code can be exploited for malicious behaviours;
- **Underlying Blockchain**: the underlying blockchain can be hacked or behave in unexpected ways;
- **User**: 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].
- **Systematic financial**: the wrapped assets used by bridges to mint the canonical version of the original asset on a new chain, can be exploited, exposing the ecosystem to systematic risk.

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

There are numerous examples of bridge attacks that resulted in multimillion dollar losses. [36].

It is noteworthy to mention how a bridge hack mainly happens from a vulnerability identified and exploited within the bridge contract, such as the Wormhole attack [37] or the Optimism smart contract bug [38]. In the remaining cases, user mistakes take place, such as in the Optimism Wintermute case [39], 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 her/his 1DLT bridge without us taking control of the bridge or custody of the assets. We offer reliable smart contracts that adhere to the community-tested security best practices, such as Optimism [40] or Polygon[41]). 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[42], Hacken[43], and Trail of Bits [44]. As part of the internal audit process, we use a smart contract bytecode verification similar to the one of Etherscan [45] and Sourcify [46]. To this end, to have the smart contracts verified, a user must share with us 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 remark that such experiments are only preliminary and do not fully account for the complexity of the experimental landscape of 1DLT (future versions of the present white paper will include a comprehensive experimental benchmark). We run experiments according to the following metrics:

1. **Total transaction cost**: the cost of sending a transaction, which is computed as:

$$Cost_{total} = cost_{transaction} + fee_{gas} + fee_{DLT} \tag{1}$$

   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. It is important to 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, generally 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:

$$Cost_{total} = cost_{transaction} + fee_{gas} + fee_{DLT} + cost_{createContract} + cost_{data} \tag{2}$$

   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 associate to the contract complexity. As of today, we do not support only the legacy format (before EIP 2930 [47])

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 $7tx/second$ [48] due to relatively small blocks and long block time. Instead, Ethereum has a short block time but tiny blocks, which results in a $15tx/second$ [49]. 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 QPQ Ethereum nodes are forwarded to the public blockchains like Hedera and Algorand. Therefore, 1DLT throughput increases 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) [50] on the Testnet as the consensus resource. We simulate the Web3 API interaction using Web3.js API [25]. We use Metamask [23] as the wallet application to verify the state of the transactions.

The experiments have been developed using the Hardhat development environment [24], as it is the de-facto standard tool for developing dapps [51].

[8]https://azure.microsoft.com/

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

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. Second, 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 (see Figure 9).

The total cost ($Cost_{total}$) for Alice is as follows: fee_{Gas} is 0.000021 *Alice_Token*, $cost_{transaction}$ is 1000 *Alice_Token*, and fee_{DLT} is 0.00051779 *HBAR*, which is 0.00000003 *Alice_Token* (assuming that 1 ETH = 1 *Alice_Token*). As such, the cumulative cost will be $1,000.00002103$ *Alice_Token* and the cumulative cost for the fees is 0.00002103 (0.056 USD). On the Ethereum Testnet (Ropsten [52]) the cumulative cost is 0.00005093 (total of 0.14 USD), while on the Ethereum mainnet, with a gas fee of 47 *Gwei*, is 0.000819 (for a total of 2.95 USD).

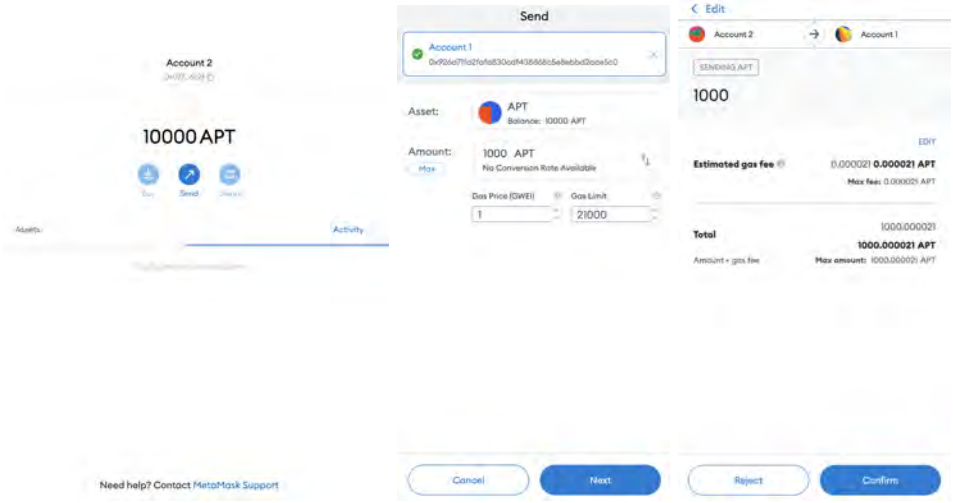

Figure 8: Setup for Send transaction from Alice to Bob

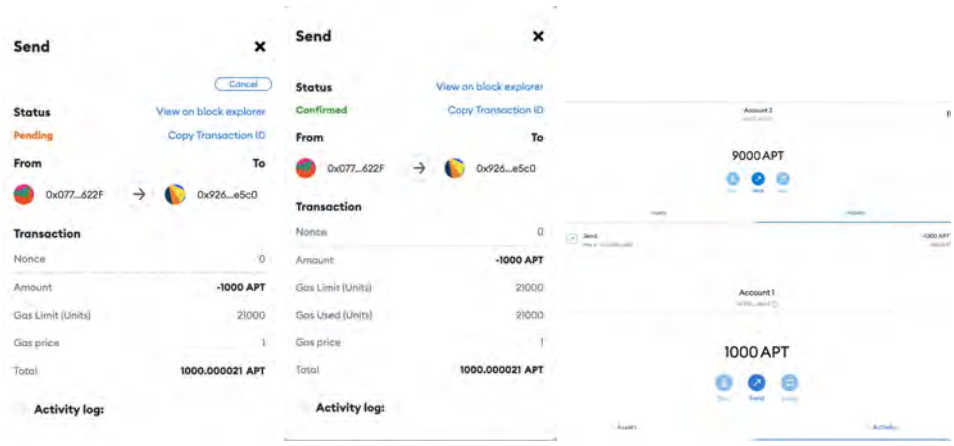

Figure 9: Send transaction from Alice to Bob state transition and account update

Transaction finality: We evaluate the consensus finality of 1DLT using a client app that generates payment transactions, and submits them to a QPQ 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:

$$Overall\_time = Generate\_Send_{tx} + 1DLT\_Finality_{tx} + Update\_wallet_{tx} \quad (3)$$

where:

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

In Figure 10, 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.

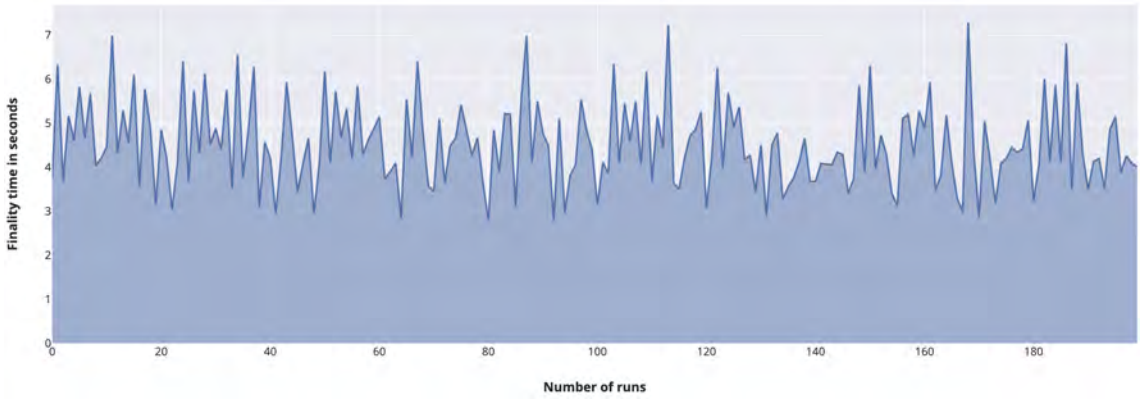

Figure 10: Transaction Finality experiment Diagram

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 to deploy the smart contract on 1DLT is 0.000013402 Alice token (suppose that 1 ETH = 1 *Alice_Token*). On the Ethereum Testnet (Ropsten) the cumulative cost is 0.0015402 (total of 1.74 USD), while on the Ethereum mainnet, with a gas fee of 30 *Gwei*, is 0.0117055 (for a total of 13.91 USD). We 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. In fact, under the hood, a setter method is implemented by sending a transaction.

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 GiB memory, and running Ubuntu 20.04. We want to underline that in this experiment, we configured CaaS to run on a small VM to prove that it is very lightweight and can achieve high performance even with this setup. Ideally, and in the production environment, CaaS will run behind a Kubernetes cluster that allows it 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.

8.1 Discussion

We now discuss the energy consumption and costs of the Ethereum network, arguing the that 1DLT offers a better trade-off between costs and performances. Ethereum currently uses a proof-of-work [53] consensus mechanism to ensure security and decentralisation. To this end, Ethereum requires massive computational power to run, with high operating costs and energy consumption [4]. In a PoW-based setting, transactions are confirmed by miners, which can add a new block to the ledger only after solving an algorithmic puzzle that entails an associated computational cost. In particular, miners compete against each others for the creation of each new block. This competition forces miners to invest in increasingly powerful hardware, creating a race to energy-hungry mining equipment.

As of now, the current Ethereum energy consumption is above 100 TWh/year [5, 54] (see Figure 11), which is roughly the energy consumption of a country like Austria. Also, the annual carbon footprint is 53.81 Mt CO [54], which is around the carbon footprint of Singapore. The average footprint for one transaction is 227.67 kWh [54], which is slightly 100 kWh (which in turn is more than the footprint of 100,000 VISA transactions [55]).

On the other side, if we take into account 10,000 miners that hash at five hashes per second, the network needs 100,000 hashes before finding a solution. Due to the PoW mechanism, only the 0.001%

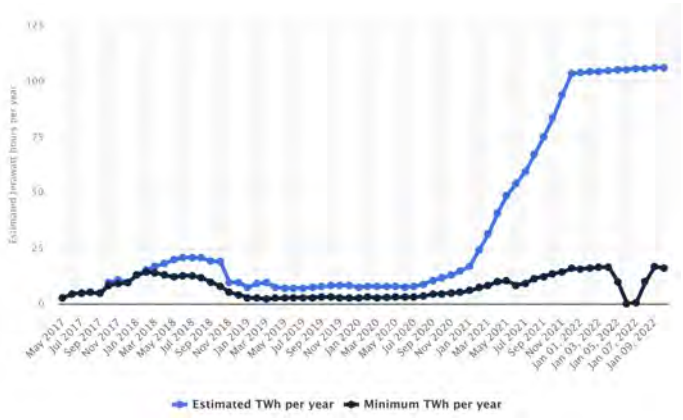

Figure 11: Worldwide Ethereum energy consumption

hashes are successful. Currently, the Ethereum network has a hash rate of 884,510 GH/s [56] and a block needs 15 seconds to be finalized [57], meaning that a miner must wait for 13,267,650 GH to gain the block's reward. Thus, the Ethereum network is currently wasting 99.999% of its computational power hashing random numbers.

Each finalized block includes 30 million Gas, which is the amount of Gas used for all the transactions in a block [58]. 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).

So far, we have discussed the computational power that is needed for keeping the Ethereum network live and running. Now, in order to estimate the cost of the Ethereum network itself (measured in gas), following [6], we consider a very basic computational task like adding two 256-bit integers. Because this operation costs 3 Gas [59] and the Ethereum network's total compute is 2 million gas/second, the Ethereum network may perform 600,000 additions per second. In comparison, Raspberry Pi 4 [60], 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 and highly energy consumption. The energy consumption will be reduced with the introduction of the Ethereum merge [61] (announced to be delivered in September 2022) as the consensus mechanism will shift from PoW to PoS. However, this change does not impact the overall price reduction, as the gas price will not be affected.

As mentioned at the beginning of this discussion, 1DLT follows a modular approach and separates the EVM-based computational layer from the consensus layer, minimising 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 works

Scaling solutions are crucial to increasing the Ethereum network's capacity in terms of speed and throughput, but they come at the cost of reduced decentralisation, increased transaction finality times, or loss of trustlessness.

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) enhance the bridge with support

for blockchains that are not compatible with EVM; (iii) the integration of 1DLT with Trusted Execution Environments; (iv) full integration in the QPQ Atomic Swap Engine (ASE), and with a QPQ proprietary wallet that will enhance the Metamask solution; (v) provision of user tools like Etherscan [62] for Ethereum or DragonGlass [63] for Hedera, to audit the status of the blockchains in the 1DLT ecosystem.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger.," *Ethereum Project Yellow Paper*, 2014.

[2] "Ethereum TPS." `https://ethtps.info/`, 2022. [Online].

[3] "Bored ape crush Ethereum." `https://www.cnet.com/personal-finance/crypto/bored-ape-yacht-club-just-broke-the-ethereum-blockchain/`, 2022. [Online].

[4] "Ethereum energy consumption." `https://ethereum.org/en/energy-consumption/`, 2022. [Online].

[5] "Ethereum energy consumption statistics." `https://www.statista.com/statistics/1265897/worldwide-ethereum-energy-consumption/`, 2022. [Online].

[6] N. Weaver, "The Web3 Fraud." `https://www.usenix.org/publications/loginonline/web3-fraud`, 2021. [Online].

[7] "Ethereum Scaling." `https://ethereum.org/en/developers/docs/scaling/`, 2022. [Online].

[8] Optimism, "Optimistic Rollups." `https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/`, 2022. [Online].

[9] "Zero-Knowledge Rollups." `https://ethereum.org/en/developers/docs/scaling/zk-rollups`, 2022. [Online].

[10] "State Channels." `https://ethereum.org/en/developers/docs/scaling/state-channels/`, 2022. [Online].

[11] "Sidechains." `https://ethereum.org/en/developers/docs/scaling/sidechains/`, 2022. [Online].

[12] "Plasma." `https://ethereum.org/en/developers/docs/scaling/plasma/`, 2022. [Online].

[13] "Validium." `https://ethereum.org/en/developers/docs/scaling/validium/`, 2022. [Online].

[14] Starkware, "Starkware Cairo." `https://starkware.co/cairo/`, 2022. [Online].

[15] Starkware, "Starkware Libs." `https://github.com/starkware-libs/`, 2022. [Online].

[16] "Ethereum client and node definition." `https://ethereum.org/en/developers/docs/nodes-and-clients/`, 2022. [Online].

[17] "Geth." `https://geth.ethereum.org/docs/`, 2022. [Online].

[18] "Erigon." `https://github.com/ledgerwatch/erigon`, 2022. [Online].

[19] "Hedera mirror service." `https://hedera.com/learning/hedera-hashgraph/what-is-the-hedera-mirror-network`, 2022. [Online].

[20] "Merkle Patricia Trie." `https://eth.wiki/fundamentals/patricia-tree`, 2021. [Online].

[21] "EVMe." `https://ethereum.org/en/developers/docs/evm/`, 2021. [Online].

[22] "EVMone." `https://github.com/ethereum/evmone`, 2021. [Online].

[23] "Metamask." `https://metamask.io/`, 2021. [Online].

[24] "Hardhat." `https://hardhat.org/`, 2021. [Online].

[25] "web3.js API." `https://web3js.readthedocs.io/en/v1.7.4/`, 2021. [Online].

[26] "Leveldb database." `https://github.com/google/leveldb`, 2022. [Online].

[27] "Sled database." `https://github.com/spacejam/sled`, 2022. [Online].

[28] "Erigon stage sync." `https://github.com/ledgerwatch/erigon/blob/devel/eth/stagedsync/README.md`, 2022. [Online].

[29] "Introduction to blockchain bridges." `https://ethereum.org/en/bridges/`, 2022. [Online].

[30] "Blockchain bridges." `https://ethereum.org/en/developers/docs/bridges/`, 2022. [Online].

[31] "The interoperability trilemma." `https://blog.connext.network/the-interoperability-trilemma-657c2cf69f17`, 2022. [Online].

[32] "Blockchain bridges classification." `https://li.fi/knowledge-hub/bridge-classification/`, 2022. [Online].

[33] "Binance bridge." `https://www.bnbchain.org/en/bridge`, 2022. [Online].

[34] "Connext bridge." `https://bridge.connext.network/`, 2022. [Online].

[35] "Hop." `https://app.hop.exchange/`, 2022. [Online].

[36] "Leaderboard of Ethereum bridge attacks." https://rekt.news/leaderboard/, 2022. [Online].

[37] "Wormhole hack." https://rekt.news/wormhole-rekt/, 2022. [Online].

[38] "Optimism smart contract bug." https://cryptoslate.com/critical-bug-in-ethereum-l2\ \-optimism-2m-bounty-paid/, 2022. [Online].

[39] "Optimism Attack." https://cointelegraph.com/news/optimism-loses-20m-tokens-after-\ \l1-and-l2-confusion-exploited, 2022. [Online].

[40] "The Optimism bridge." https://ethereum.org/en/developers/tutorials/optimism-std-bridge-annotated-code/, 2022. [Online].

[41] "The Polygon-Ethereum Bridge." https://docs.polygon.technology/docs/develop/ethereum-polygon/getting-started/, 2022. [Online].

[42] "CertiK." https://www.certik.com/, 2022. [Online].

[43] "Hacken." https://hacken.io/, 2022. [Online].

[44] "Trail of Bits." https://www.trailofbits.com/, 2022. [Online].

[45] "Etherscan smart contract verification." https://etherscan.io/verifyContract, 2022. [Online].

[46] "Sourcify." https://docs.sourcify.dev/docs/intro, 2022. [Online].

[47] "EIPS 2930." https://eips.ethereum.org/EIPS/eip-2930, 2022. [Online].

[48] "Real time Bitcoin finality." https://statoshi.info/d/000000006/transactions?viewPanel=6&orgId=1, 2022. [Online].

[49] "Real time Ethereum finality." https://ethtps.info/, 2022. [Online].

[50] "Hedera Consensus Service." hedera.com/consensus-service, 2021. [Online].

[51] "Solidity report 2021." https://blog.soliditylang.org/2022/02/07/solidity-developer-survey-2021-results/, 2022. [Online].

[52] "Ethereum testnets." https://ethereum.org/en/developers/docs/networks/, 2022. [Online].

[53] "Ethereum Proof of work." https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/, 2022. [Online].

[54] "Ethereum energy footprints." https://digiconomist.net/ethereum-energy-consumption/, 2022. [Online].

[55] "Ethereum energy consumption compared to VISA." https://www.statista.com/statistics/1265891/ethereum-energy-consumption-transaction-comparison-visa/, 2022. [Online].

[56] "Ethereum hashrate chart." https://etherscan.io/chart/hashrate, 2022. [Online].

[57] "Ethereum statistics." https://ethstats.net/, 2022. [Online].

[58] "Ethereum gas." https://ethereum.org/en/developers/docs/gas, 2022. [Online].

[59] "Ethereum opcodes." https://ethereum.org/it/developers/docs/evm/opcodes/, 2022. [Online].

[60] "Raspberry datasheet." https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf, 2022. [Online].

[61] "The Ethereum merge." https://ethereum.org/en/upgrades/merge/, 2022. [Online].

[62] "Etherscan." https://etherscan.io/, 2021. [Online].

[63] "DragonGlass." https://app.dragonglass.me/, 2021. [Online].

Simone Bottoni is a Research Engineer at QPQ. He is also a PhD student in Computer Science at Insubria University, Varese, Italy, and holds an MSc in Computer Science from Insubria University, Varese, Italy.

Anwitaman Datta is a Senior Scientific Officer at QPQ. He is also an Associate Professor at the School of Computer Science and Engineering in NTU Singapore. He holds a PhD in Computer and Communication Sciences from EPFL, Switzerland, and a B. Tech in Electrical Engineering from IIT Kanpur, India.

Federico Franzoni is a Research Engineer at QPQ. He holds a PhD in Information Technology from the University of Pompeu Fabra, Barcelona, Spain, and an MSc in Computer Science from Sapienza University, Rome, Italy.

Emanuele Ragnoli is the CTO at QPQ. He holds a PhD in Systems Theory from the Hamilton Institute, NUI Maynooth, Ireland, and an MSc in Mathematics from Milan University, Italy.

Roberto Ripamonti is a Research Engineer at QPQ. He holds an MSc in Computer Science from Insubria University, Varese, Italy.

Christian Rondanini is a Research Engineer at QPQ. He holds a PhD in Computer Science from Insubria University, Varese, Italy, and an MSc in Computer Science from Insubria University, Varese, Italy.

Gokhan Sagirlar is a Research Engineer at QPQ. He holds a PhD in Computer Science from Insubria University, Varese, Italy, and a BSc in Computer Science from Ege University, İzmir, Turkey.

Alberto Trombetta is the CSO at QPQ. He is also an Associate Professor of Computer Science at the Department of Theoretical and Applied Sciences of Insubria University. He holds a PhD in Computer Science from Turin University, Italy, and an MSc in Computer Science from Milan University, Italy.