Micropayments Interoperability with Blockchain and Off-Chain Data Store to Improve Transaction Throughputs

Azmi Amiruddin(✉)

TU Berlin, Berlin, Germany
azmiruddin@tu-berlin.de
https://www.tu-berlin.de

Abstract. Layer2 (L2) techniques have emerged as promising scalability solutions to improve the quality of blockchains services, particularly to increase transaction throughputs. We instantiated a multilayered approach for our L2 scalability solution, composed of a node server, an off-chain data store, smart contracts (for state channel and payment network), and a REST API as an oracle. In the first release of our application, we demonstrated how state channel, payment network, and off-chain data stores allowed us to critically examine how scalability can be achieved by incorporating a state channel, payment network, and off-chain data control. We achieve flexibility and interoperability using proven standards (e.g., ERC-20 and EIP-1474), conducting a multivocal literature review, integrating existing approaches using a payment channel framework, and proposing efficient process execution to reduce on-chain transaction overheads using an off-chain data store mechanism. In return, the proposed solution can reduce on-chain transaction overheads by combining a novel framework that shifts interoperability among blockchains, state channels, payment networks, and off-chain data stores.

Keywords: Blockchain · Layer two · Off-chain · Scalability

1 Background

According to Forbes [7], the six largest financial firms and three technology giants have started to invest in their blockchain ecosystem for widespread adoption. This report is aligned with Gartner's research [2], which predicted that 90% of current blockchain adoptions would require a "technology refresh" to remain competitive and avoid obsolescence. Melanie Swan in [31] placed a fundamental blueprint for the blockchain technology hype. Blockchains' advantages are more than technological innovations and economy; they extend into the political system through digital voting [31]. Blockchain technology offers the opportunity to develop new business and trust models, so the blockchain during the *peak of inflated expectations* is one of the game changers that will necessarily involve new technical foundations and more dynamic ecosystems [26].

S. Paiva et al. (Eds.): SmartCity360° 2021, LNICST 442, pp. 521–549, 2022.
https://doi.org/10.1007/978-3-031-06371-8_34

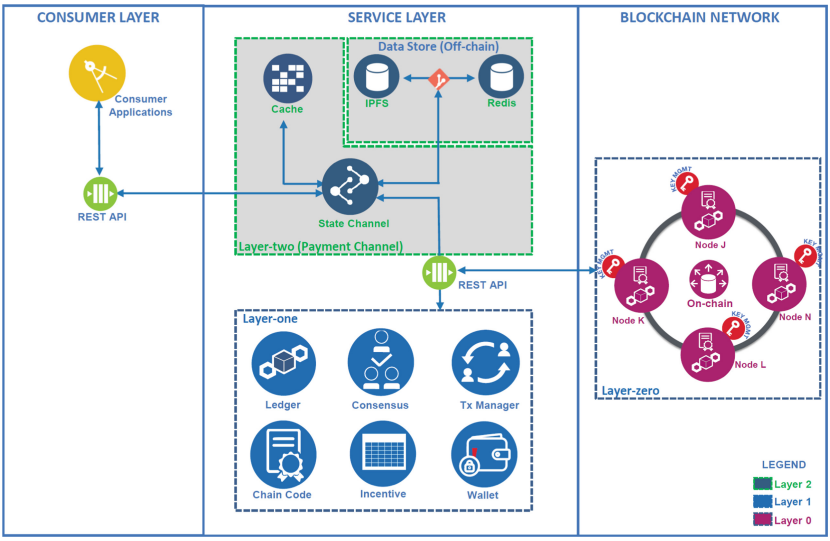

Fig. 1. Blockchain layer

Despite the hype cycle of blockchain implementation in all sectors, the processing time to handle transactions is largely underperformance as compared to that in a legacy system [17]. Research in [17] classified blockchain scaling solutions based on (i) a consensus algorithm, (ii) sharding technique, and (iii) side-chain solutions. In addition, in [17], highlighting the drawbacks of the consensus algorithm indicates modifying one of the core components of a blockchain layer one (L1) while already in use, which creates major unplanned issues, such as lack of backward compatibility, clearly impeding their practical implementation. Changes in L1 might even result in multiple forked systems (e.g., different forked systems are implemented in the blockchain test network, such as Rinkeby using proof of authority (PoA) and Ropsten using proof of work (PoW)). In [17], one promising solution was classified as an L2 scalability solution. L2 components operate independently from L1 components. As a result, some transaction processing within layer 0 (L0) moves outside the blockchain network (off-chain computation).

In this research, we combine state channel framework and payment network protocols, also referred to as an L2 scalability solution. In contrast to the previously mentioned L1 solutions, L2 protocols scale blockchains without affecting the L1 core components. L2 protocols, which are presented in Fig. 1, enable users to save block transactions in the preferred data store technology and then perform off-chain transactions through private and encrypted peer-to-peer (P2P) messages, rather than broadcasting each transaction to the leading blockchain network. This optimization reduces block size and transaction load on the underlying blockchain while remaining completely backward compatible. In [17], the theoretical transaction throughput is only bounded by the data store, network service-level

agreements (e.g., bandwidth, density, and latency) of the involved participants. Off-chain transaction confidentiality, integrity, and availability can be guaranteed through collateral allocation, such as in payment channel designs [13,30] or in providing delayed transaction finality in commit-chains proposal [20].

1.1 Problem Description

Common problems arise when transferring transactions between merchants and buyers. A trusted intermediary is a must in agreements to enforce the terms, but it could be malicious. How is the data transaction managed if the participant is not connected to the system? How to manage when the transaction is partially not submitted to the ledger directly? We devised solutions to mitigate both issues by implementing a state channel and payment network using a data store. We reviewed three significant problems related to scalability issues due to blockchain adoption according to [31,33].

1. **Throughput**. The public blockchain network has limitations with transaction throughput as it processes only 1 tps, with a theoretical current maximum of 20 tps for the Ethereum network [31,33]. One method to handle a higher throughput is increasing each block size, but it causes block size and blockchain bloats issues. Comparison metrics in permission and consortium blockchain network are RedBelly (660,000 tps) [10], R3 Corda (1678 tps; 15,000 tps peak) [29] and VISA networks (1,500 tps typical; 7,000 tps peak) [18].
2. **Latency**. Each blockchain transaction block takes 10 min to process and confirm a transaction [31]. Here transaction timeout should be allotted for sufficient security, particularly for large amounts, as the waiting time needs to be longer to override the financial loss of a double-spend threat.
3. **Block size and data placement**. Ethereum blockchain takes 117 GB, and it has grown by 48 GB in the last year[1]. Downloading Ethereum full node takes hours or even days. If 2,000 tps increase the throughput to VISA standards, the result is 1.42 PB/year or 3.9 GB/day [31]. Accordingly, at 150,000 tps, the blockchain would grow by 214 PB per year. The blockchain community refers to the size issue as a "bloat". To scale blockchain to a *plateau of productivity*, it would need to be large and distributed, likely impacting transaction fees like on-chain data costs.

With the issues above confronting the blockchain public network, we identified an opportunity to develop a new application with data store placement to keep the blockchain usable and storable (at much larger future scales) while maintaining its confidentiality, integrity, and availability. Exposing application programming interfaces (APIs) to external consumers, such as those from L2 solutions, which facilitate automated calls to the entire blockchain network, is an

[1] Data snapshot as of 30 December 2020, obtained from https://blockchair.com/ethereum.

innovation to address blockchain "bloats" and make data more accessible. Some operations include obtaining address balances, changing balances, and pushing notification service when new transactions state or blocks are created on the network. Other operations are web-based block explorers (e.g. Etherscan[2] or Blockchain explorer[3]), decentralised applications (DApps), mobile digital wallets, middleware software that allow seamless integration and replica placement for off-chain data store (e.g. Chainlink solution [3]). Finally, developing an interoperable framework to address new business requires an understanding of how systems operate to create opportunities that influence blockchains to minimize the challenge on the *plateau of productivity*.

1.2 Research Questions

Blockchain adoption seeks stable guidelines for successful implementation and improvement in systemic qualities during the plateau of productivity stage. Our primary research objective is to achieve scalability in Ethereum blockchains. Moreover, driven by this research objectives in guidance of [1,15–17,21], our research questions are as follows:

1. RQ 1: What is state of the art in blockchain scalability models, and how their design patterns look like?
2. RQ 2: How can the architectural approach integrate data store and state channel technique to improve the QoS of the existing blockchain implementation?
3. RQ 3: How can a datastore from [21], state channels [13] and payment network [30] be integrated to improve scalability (increasing transaction throughput) for the current blockchain implementation?

1.3 Contributions

1. An experimental approach is provided by the architecture of incorporating a blockchain into a data store and payment channel solution with off-chain control to improve scalability problems like block size and transaction throughput. A framework to integrate with the baseline work [21] was selected with payment channel technique from [13,30]. Minimizing system quality effect and improvement on the existing payment channel architecture are focused.
2. A practical off-chain technique is built upon a novel multi-tier application based on (i) state channel solution [13,30] and (ii) persistent mechanism, which relies on a distributed data store implemented using Redis and IPFS [21].
3. An application that moves these roles into a layered architecture with a separation of concern for the data layer, business process, resource adapter, and user interface into a separate component-based application is introduced and developed in a method called resource separation (refer to "Single Responsibility Principle").

[2] Ethereum blockchain explorer https://etherscan.io/.
[3] Bitcoin blockchain information https://www.blockchain.com/explorer.

Finally, in the future, we will have an architectural goal to expose other L2 scalability solutions (e.g., probabilistic micropayment [6] and incentive mechanism [1]).

1.4 Related Work

Wattenhofer and Decker [11] created an off-chain channel termed blockchain L2 scalability. Some studies define protocols for blockchain L2 scalability and channel networks. The off-chain protocol for the Ethereum network, is covered in (i) Connext [9], (ii) KChannels [19], (iii) Go-Perun [13] and (iv) Raiden [30]. Moreover, in Appendix 1, we provide an overview of current L2 constructions, distinguishing between two-channel techniques: (i) state channels from Go-Perun [13], which support off-chain payment interactions, and (ii) payment networks from Raiden [30], which support off-chain arbitrary interactions.

There are particularly relevant white literature and GL that aim to offer blockchain L2 scalability solutions like the Lightning Network [28]. On the industrial project, a white paper for L2 scalability is presented in Counterfactual [8], and Celer Network [12]. GL related to these projects puts more emphasis on the "engineering-oriented" approach and product information sheet. Thus, it does not discuss the characteristics of the full-state channel procedure. Moreover, most of this work covers all networks in blockchains, including Bitcoin and Ethereum. The most useful literature for this research was described in [13,30], but it did not highlight the solution on the data store approach. Another proposal that works inside the Ethereum network is presented in [23] with release name Sprites, and its extensions Pisa [22] on building multi-party ledger state channels did not discuss the clear data store approach. A group of parties can initiate a multi-party ledger state channel as per [22,23] and a disagreement between the parties is resolved on-chain. Hence, the solutions from [22,23] are not listed by Ethereum community development[4].

Table 1 presents the comparisons of the proposed research work with the existing solutions.

Table 1. Comparison of proposed work with existing solutions

Proposal	Solution
Connext [9]	TypeScript, JavaScript, Extend Hash-Time-locked-Contract, Distributed Balanced Routing
KChannel [19]	TypeScript, Meta-channels, Replaced-by-Incentive
Perun [13]	Go, Replaced-by-Version, Virtual Channel
Pisa [22] and Sprites [23]	Python, Replaced-by-Version, State Channel
Raiden [30]	Python, Hash-Time-locked-Contract, Replaced-by-Revocation, Machine-to-Machine Payment, State Channel
Proposed Work	Java, Datastore, State Channel, Hash-Time-locked-Contract, Replaced-by-Revocation, Replaced-by-Version

[4] Ethereum L2 Scaling https://ethereum.org/en/developers/docs/scaling/state-channels/.

There are two major distinctions in our work: first, [13,22,23,30] did not support flexible data store mechanisms, and as a result, interacting with the blockchain becomes necessary to establish and close state channels. Second, the solution suggested in this research completely supports the simultaneous execution of multiple contracts within a single channel, whereas [13,22,23,30] focused on the off-chain execution of merely a single contract with additional evaluation on the practical aspects of state channels. We suggest a different channel state solution in this research. Defining a formal data store mechanism is also a primary objective of this research, which was adopted in [21], and our contribution aims at improving the transaction throughput in Ethereum networks, using state channels and a multi-layered architecture that has light components to be plugged in other ecosystems, such as the Internet of Things (IoT).

2 System Design

As an end-to-end solution that can be developed and integrated on existing or future blockchains, our state channel application includes a clean layered architecture that decouples the sophisticated L2 scalability into modular components. The combination of this architecture simplifies the software development life cycle (SDLC), allowing each independent service to evolve inside a continuous integration and delivery pipeline (CI/CD). A well-designed layered architecture should have open interfaces that will allow and encourage different layer implementations as long as they support the same cross-layer interfaces. Moreover, each component only needs to concentrate on its functionality (componentization via services).

One of our architectural goals is to enable state channel techniques, off-chain data control, and object interaction logic that can be used across multiple blockchain networks. Our state channel application aims to create a blockchain-agnostic interoperable platform that can run in the Ethereum networks. Therefore, we adopted a standard off-chain datastore schema, EIP-55 [4], and EIP-1474 [27]. It is also aimed to provide innovative solutions on blockchain L2 and improve scalability for blockchain technology adoption. Our target architecture builds a multi-layer model by adopting a data store engine from [21], state channel technique in [13] and payment network from [30]. Our target multilayered architecture, as shown in Fig. 2, is made up of the layers described below, in bottom-up order.

1. **Data Store**. Fog computing, a novel technology that combines cloud computing resources, allows easy movement of computational infrastructures. Using a prominent data store engine as proven in [21] enables our application to have portability and scalability features simultaneously.
2. **Channel Node**. Our application uses the state channel technique from Go-Perun [13], and Raiden network [30], which allowed us to reduce the scalability challenges that major public blockchain platforms face (e.g., Bitcoin and Ethereum).

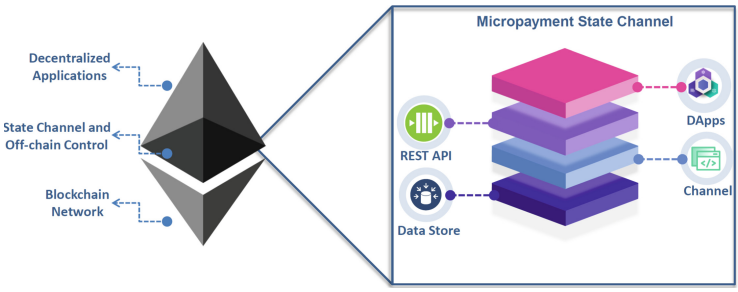

Fig. 2. Target architecture

3. **REST API**. Communicating via lightweight service mechanisms (HTTP resource API) enables the accessibility of consumer applications and semantic interoperability that can be achieved through models of information.
4. **DApps**. The front-end framework, which runs a GUI for off-chain-enabled applications, is accessed directly by the consumer. Our DApp interfaces, namely, React Native framework[5] and JSON server-client component[6], allow us to develop a clear user interface and allow SoC for each component.

The following section will briefly discuss the system functionality based on the layered architecture illustrated in Fig. 2.

2.1 Off-Chain Data Control

We modified the RDF store engine [21] by adding additional logic for saving off-chain transaction data and data portability. Store data come from transaction log where wrapper invokes and returns the standard EIP-1474 message specification [27].

Channel Object. Before the pointer requests a state modification triggered by the consumer application, the response is dispatched to the mediator controller. Next, the *TransactionSave* persists the raw data into the *DataStore* object. This allows the *DataStore* function to record all incoming state channel messages into *RedisBufferLayer* and *IPFSPhysicalLayer*, and all event processes for state channel applications are fully synchronized into Redis and IPFS. The data store process will be executed, as shown in Fig. 3, if the channel participants initiate a state event that is received from an Ethereum transaction. Subsequently, the data input will be parsed into a raw transaction message from the blockchain (JSON data block), and the raw message will be converted into an appropriate plain object that will be parsed into the key-value store.

The data placement feature makes our application different from Go-Perun and Raiden because it allows "on-the-fly" compilation to manage the data store

[5] Develop native apps https://facebook.github.io/react-native/docs/getting-started.
[6] JSON DB inside clients browser https://www.npmjs.com/package/json-server.

in case of disputes, crashes, and irregular shutdowns. The shortest path for datastore processing is shown in Fig. 3 and the detailed information flow for channel object creation is presented in Fig. 4.

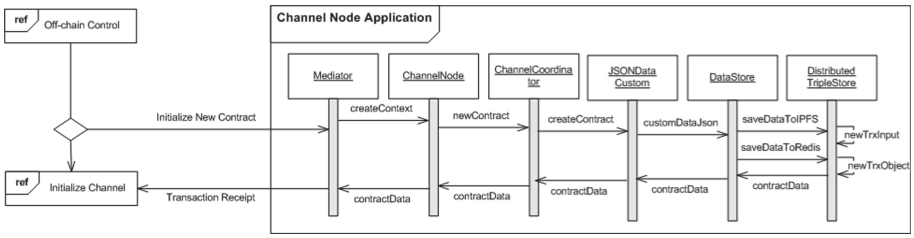

Fig. 3. Data store process

After the data store process starts, the channel object will automatically parse the blockchain log messages (JSON data) and transaction events into our key-value store with the help of *Converter*, *ChannelStatusResponse* and a Web3J wrapper.

1. **Channel Initialise**: initiate to open a channel based on the predefined channel contract address
   a. address: string EIP-55 [4], which encoded the address containing the identifier of the channel[7].
   b. status: response of the contracted event.
   c. transferred: number of assets to be exchanged.
   d. current one: current block transaction count.
   e. synced once: synchronized block transaction with Ethereum.
   f. stateRoot: string containing the EIP-55 encoded condition of the channel represented by a string from the state root. The possible events are as follows: (i) Open: the channel state is created, and tokens are tradeable. (ii) Settlement: the channel has been utilized by a participant and settled. (iii) Closed: the channel has been requested to be destroyed by a participant.
2. **Channel Settlement**: settle the open channel after the successful approval of channel participants deposits
   a. sender: string containing the EIP-55 encoded sender address of the partner with whom we have opened a channel.
   b. receiver: string containing the EIP-55 encoded recipient address of the token network the channel is part of.
   c. auditor: string containing the EIP-55 encoded auditor address to watch the transaction from dishonest behavior.
   d. minActive: minimum number of participants required to open the channel (the possible number is 1).

[7] For EIP-55, we implemented *Keccak256* in *ChannelLibrary* contract.

e. maxActive: maximum number of channel participants (implement using*@ConfigurationProperties* with predefined number is 50).
f. deposit: number of assets to be placed during a request to open the channel.
g. closeTimeout: channel automatically closes if the number of participants fails to deposit the token.
h. settleTimeout: string containing the EIP-55 encode blocks time required to be mined from the time the transfer has been locked.
i. auditTimeout: blocks time that is required to be disputed during transaction reversal.
j. closeBlocksCount: block number when the channel requests to close.
k. stateTransition: string containing the EIP-55 encoded transition state. Once the root state has been established, possible state transitions are as follows: (i) Deposit: the channel that has been opened requires to be funded by participants. (ii) Deposit approved: the deposit transfer has been approved. (iii) Active: the channel is open and can accept participants to join the channel. (iv) Dispute: there is a fraudulent transaction where one of the participants automatically changes the state. (v) Transfer lock: the transfer has been locked by the recipients. (vi) Transfer unlock: participants request the asset to be settled in their wallet. (vii) Shutdown: the participant requests to destroy the channel. (viii) Destroy: the channel is automatically closed.

3. **Channel Transfer**: update the open channel with the transaction between participants
   a. channelId: string containing the identifier of the channel.
   b. transferred: string containing the EIP-55 encoded transfer identifier from Ethereum sends the transaction event (transaction root).
   c. value: the amount of transaction to-be transfer (possible value is less than the token value).
   d. locked: mix hash key to lock the transfer; the EIP-55 encoded value from Ethereum sends the transaction event (transaction root).
   e. signature: transaction hash from channel participants.

The transaction events converted from Ethereum messages are JSON file types and taken care of by the object *TransactionSave*. Then, the results will be saved into a JSON file, handled by the object *TransactionOutputOffchain, ObjectFileTransactionOffChain*. Once the data have the results fixed, the object *SerializeUtils* will serialise and de-serialise the JSON file and insert it into the Redis store with a key. Figure 4 shows the transaction processing on this procedure.

Off-chain Data Lifecycle. Our system starts with the extraction and transformation of raw messages from Ethereum transaction events into our key-value store with the help of plain Java object conversations. The data were used for off-chain computations, such as reading, updating, insert and deleting transaction records. The processing flow of the off-chain data lifecycle presented in Fig. 5 can be briefly explained as follows:

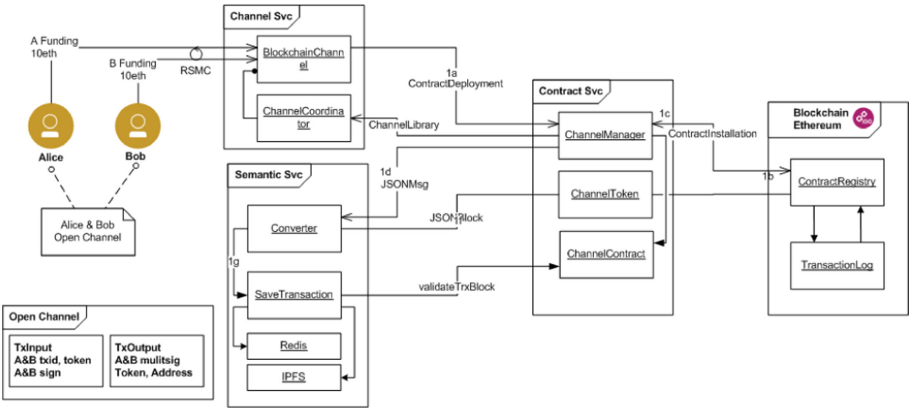

Fig. 4. Information flow (open channel)

- [1a] Channel participants have to deposit tokens into the blockchain and *BlockchainChannel* will handle their request using contract deployment. In this flow, Alice and Bob agreed to deposit 10 ethers to open the channel.
- [1b] A request for contract deployment is received by the *ContractManager* inside the channel service component and then parsed into a specific contract wrapper that will communicate with the Ethereum network (e.g. *ChannelContract* or *ChannelToken*). Deployment of *ChannelContract* and *ChannelToken* is required in case of off-chain computation.
- [1c] The contract deployment will continue to be installed inside the EVM, the *JSONData* transaction response for this request will be used for our data, and the response will be saved into the key-value store. Inside the data store component, the *Converter* object then converts the data block into our predefined data format.
- [1d] Once the transaction has been saved into Redis and IPFS, and the transaction data block will be validated and returned to the consumer. Once the response has been received, the *ChannelCoordinator* will change to the *run/start* mode to watch the state transition of the channel lifecycle and return the transaction event to the *BlockchainChannel*.
- [1e] After the *ChannelCoordinator* starts in the run mode, and every state change will be processed to get the event. For each of the *StateTransition* events, *OutgoingChannelState* will record the conversation and convert the transaction events and trigger the appropriate data persistent into Redis and IPFS.

During the request to create a channel, the data will be dispatched into the Ethereum network: participant addresses and deposits will be placed in the token network. After deposit approval, the response will be sent back to consumer applications, and the token can be utilised to initiate any off-chain transaction. Before the response message is sent back to the consumer, a state change is dispatched to the *StateTransition* events for processing to write our data into

a key-value store called the *TransactionOutputOffchain* and *ObjectFileTransactionOffChain*.

2.2 Channel Design

We built our channel node using Java programming techniques (JDK v1.8)[8] and utilised Web3J[9] to wrap modified contracts from Go-Perun [13] and the Raiden Network [30]. We also utilised the open-source library from Papyrus Network [25], and Blockchain Thunder [24] to build channel-node servers and deploy smart contracts "on-the-fly" when a channel has been established (e.g., in a token reload or channel destroy).

We encapsulated all state channel resources in a layering fashion, as presented in Fig. 2. All the components inside the channel node were deployed using Java Archive (JAR library) and distributed into the application server as the resource library to support DApps. The overall smart contracts design will be briefly presented in the following subsection, and the node server application will be presented further in subsection channel node.

Smart Contracts. Our smart contracts are written using the Ethereum solidity language. Each contract must be deployed before a channel node can be used. However, because this is a complete system development for off-chain computations, we must deploy all of the contracts into the Ethereum test network[10]. Our state channel framework uses interconnected intelligent contracts to define the on-chain logic for channel transitions and off-chain computation for settlements and withdrawals. In our current release[11], we have six smart-contract that will support our application node. Hence, we limited our discussion to highlight the smart-contract that is mandatory to support channel node server to (i) *ChannelManager*; (ii) *ChannelContract*; (iii); and (iv) *ChannelApi*.

1. **Channel Manager**
   The *ChannelManager* is responsible for managing the root state for the channel application. It also needs to ensure that the interface binding for *SCHToken* is in sync with the token contract for each channel that is successfully open. As shown in Rinkeby test network[12], we show that related topics for channel creation also bind to specific token channels and addresses.
2. **Channel Contracts**
   The *ChannelContract* ensures recording of a transition state into the channel-node repository. Because the main part of the communication is off-chain, it will only contact the *ChannelContract* if a transition occurred during the

[8] Java documentation https://docs.oracle.com/javase/8/docs/.
[9] Develop on Ethereum with the JVM http://docs.web3j.io/latest/quickstart.
[10] Contract Overview https://rinkeby.etherscan.io/address/0x91db6dce5c2584605d7.
[11] State Channel Application https://github.com/azmiruddin/state-channel.
[12] Contract manager deployment in the Rinkeby network. https://rinkeby.etherscan.io/tx/0x6824.

transaction from channel participants (e.g., open, renew deposit, approved deposit, destroy and close). As we are demonstrating the resiliency of componentisation via services, the *ChannelContract* contract also extends *ChannelLibrary*. It will receive transaction messages (EIP-1474 standard) from the *ChannelLibrary* contract and supply the record to *ChannelContract* with a JSON message. The *ChannelContract* records transaction events related to state transition, which will sync with the channel node via the *ChannelCoordinator*. Client states managed by *ChannelContract* are:

- *ChannelNewBalance* looks after the reload/renew amount of deposit that has been proposed by the channel participants.
- *ChannelCloseRequested* is executed if one of the participants request to gracefully shutdown the channel.
- *ChannelDestroy* is responsible to force-close the channel if the participants cannot transfer the deposit.
- *TransferUpdated* operation updates the transaction.
- *ChannelSettled* operation will cover the operation after the deposit has been transferred and locked inside the token network.
- *ChannelAudited* if there is a dispute transaction, then the reversal will be triggered by the event *ChannelAudited*.
- *ChannelSecretRevealed* will take the responsibility of storing the block height at which the secret was revealed in an off-chain transfer. In collaboration with a watching service *ChannelAudited*, it acts as a integrity measure, allowing all channel to withdraw the transferred tokens. This event also unlocks the transfer.

We also conveyed to fork operations from *ChannelContract* and *ChannelLibrary*, which will improve the processing time and lower gas fees during contract compilation and deployment.

3. **Channel Token**
   SCHToken is a ledger contract that manages channels and deposits on-chain. Each channel will employ a specific address (token network address) that works with a particular ERC-20 token and manages channels between participants. The *SCHToken* deployment manages the deposited tokens and is the main point of contact for any on-chain operations for channel-node applications. Inside *SCHToken*, we also implemented a wallet feature with the capability to hold or receive withdrawal amounts from transactions. The wallet itself is directly associated with the token owner, and it will hold an amount related to the maximum number of deposits submitted during the open channel initiation. We demonstrated the feature of the token network in token test transactions from the Rinkeby test network[13].
4. **Channel API**
   Channel-node applications directly interact with the *ChannelApi* contract, which defines two methods: (i) *applyRuntimeUpdate*, which will be assigned to the channel having state transition, and (ii) *applyAuditorsCheckUpdate* to monitor the off-chain transactions of participants. The *applyAuditorsCheckUpdate* method is also responsible for the dispute and reversal phase during

[13] Sample state channel transition https://rinkeby.etherscan.io/address/0x91db6.

off-chain transactions, and it is assumed to revert if any app-specific check fails. The *ChannelApi* will apply state transitions to every new channel that is in a settled state. In our implementation, the communication facilities from the client application (channel participants) with the smart contract were provided by the Web3J library [32]. The operation was used by the *applyRuntimeUpdate* method of the *ChannelApi* contract to forward the operation to the *ChannelManager* contract.
Furthermore, the lifecycle of *applyRuntimeUpdate* and *applyAuditorsCheckUpdate* functions is similar to that described in [17], for the watching service mechanisms. The aim of *ChannelApi* is to demonstrate how a client request into specific contracts can communicate with each other, and it has a watchtower committee (*ChannelApiStub*) for the channel event itself (watching service for the specific transaction state). The detailed execution sequence of the watchtower committee is shown in Appendix 2.

We also deployed two additional contracts to support the main smart-contract deployment: (i) *ChannelApiStub* that will create the interface stub to other contract and (ii) *ChannelLibrary* that will support the *ChannelContract*. In our implementation, the interface call facilities provided by Web3J for the client node application with the node server and gRPC[14] for server-to-server communication were used by the *loadPredeployedContract* method of the *ChannelManager* contract to forward the state operation to the *ChannelApi* and *SCHToken* contracts. Intuitively, the function calls the *loadPredeployedContract* to instantaneously load the token and *channel_api* functions of the *ChannelManager* contract. The following sections will examine the integration procedure between the channel node and channel contract.

Channel Node. The channel node serves as a container server for all channel objects used in our state channel application. Our channel-node server has a hybrid implementation, such that it can be deployed using an open-source application server, e.g., Apache Tomcat[15] or JBoss WildFly[16]. The channel-node server can also stand alone, for example, using the legacy Java archive application runtime or implementing the Spring Boot framework[17].

Our application demo, as shown in Fig. 5, proceeds as follows: If the *NodeServer* is started, then the channel client application can begin transactions. The channel request will be directly routed to the OutgoingChannelCoordinator and *IncomingChannelManagers* within the node server, allowing the client to open channels with other participants within the same channel address and close channels, transaction inquiries, and make deposits or withdrawals. Additionally, *ChannelJoinImpl* enables channel composability for new participants to join the channel, transfer deposit and close off-chain over the state-channel

[14] gRPC and protocol buffers https://grpc.io/docs/.
[15] Apache Tomcat v8 https://tomcat.apache.org/index.html.
[16] JBoss managed application runtime https://docs.wildfly.org/.
[17] Spring boot execution runtime https://spring.io/projects/spring-boot.

network intermediaries using *OutgoingChannelCoordinator, IncomingChannelManagers, BlockchainChannel* (the sequence process is depicted in Appendix 3). When the node server is starting, the detailed execution sequence in Fig. 5 will proceed as follows: the node client application will request *getEndpointUrl*, and once the endpoint is established, the subsequent event *checkAddress* will determine whether the contract has been deployed with a valid return address from *ChannelApiStub, ChannelLibrary, ChannelManager, EndpointRegistryContract,* and *SCHToken*. If one of the check addresses does not match the address of an Ethereum pre-deployed contract, the node server will throw an exception during the startup process.

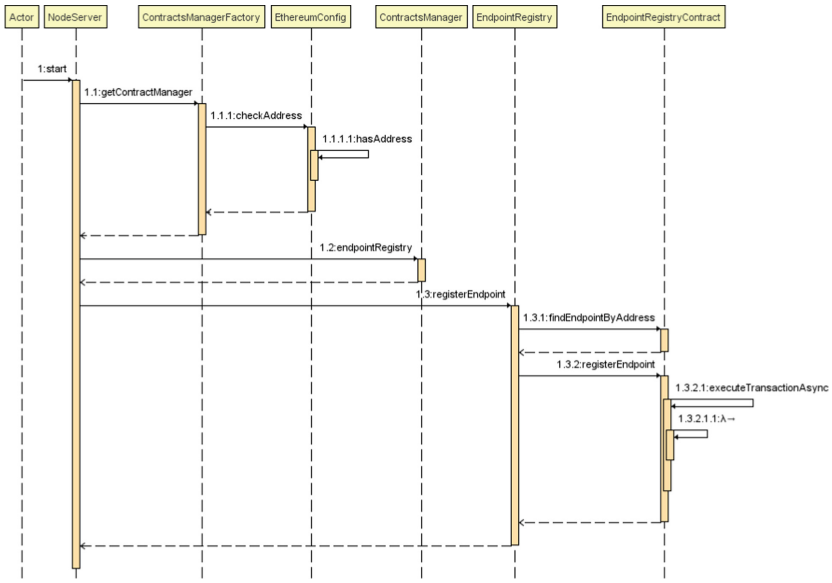

Fig. 5. Channel node start process

- *NodeServer* assumes that the participants are ready to start off-chain transaction.
- *ContractManagerFactory* validates and checks the addresses deployed into the blockchain.
- *EthereumConfig* checks if the address value in the properties is equal to the pre-deployed contract in the blockchain.
- *ContractsManager* assigns the node server to a specific client and then *EndpointRegistry* returns the binding address to the node server.

Once the execution sequence in Fig. 5 is successfully processed, the next *OutgoingChannelState* and *BlockchainChannel* will handle all state modifications and transfer the process execution to the *makeTransitions* operation that is demonstrated in Fig. 6. If the state is opened, then the channel participants

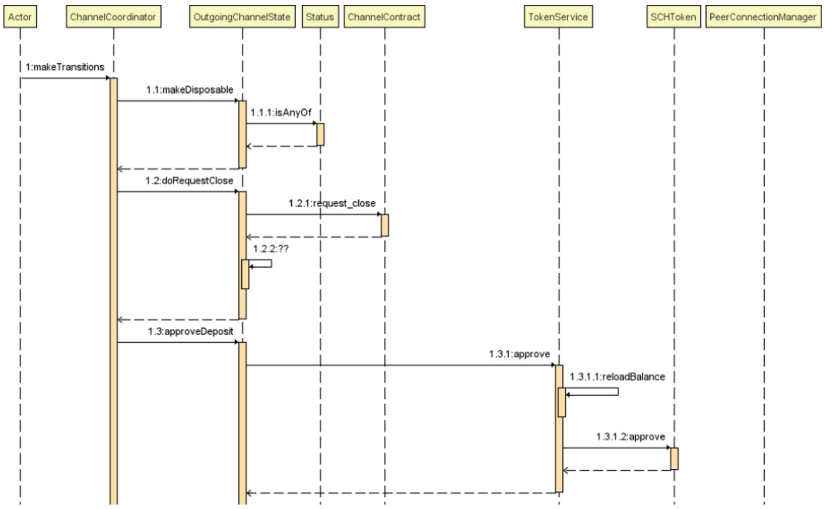

Fig. 6. *ChannelCoordinator* object starts to initiate the state transition

are directly updated off-chain between the two connected end-users, or other participants are allowed to join the channel with a predefined token address encapsulated inside the *ChannelManager*.

Once the execution sequence in Fig. 5 is successfully processed, the next *OutgoingChannelState* and *BlockchainChannel* will handle all state modifications and transfer the process execution to the *makeTransitions* operation that is demonstrated in Fig. 6. If the state is opened, the channel participants are directly updated off-chain between the two connected end-users, or other participants can join the channel with a predefined token address.

- In (1) sequence flow, the *ChannelCoordinator* assumes that channel participants are ready to start the procedure for off-chain transactions. Once the start event is the trigger, the *makeTransitions* will enable the sequencing process for state channel transitions.
- In (1.1.1) activities and subsequent process, the *OutgoingChannelState* validates and checks the address deployed on the blockchain network. The contract has any value of the following states: open, shutdown, destroyed and settled.
- In (1.2) process and subsequent activity demonstrated that the channel would be destroyed automatically if the deposit amount agreed to transfer is not accepted by one of the channel participants.

In our smart contract design, we briefed *ChannelContract* and *ChannelManager* objects that will have the main responsibility of logging the event for state transitions. The two smart contracts will be leveraged in our node server application and during the open state. Our transition states are valid for events: transfer deposit, reload deposit, approved deposit, and destroy channel. The details of

the sequence diagram executed by *SCHToken* are presented in Appendix 4. The state interaction execution in Appendix 4 was assumed successful. Here the lifecycle of a state channel is divided into three stages, as shown in Appendix 4: open, update and close.

1. **Open**. If participants want to open the channel, they can transfer some deposits to form the channel. If the other participants cooperate, this process can be done quickly in a single transaction. Suppose the other participants are not available or disagree to cooperate. In that case, the tokens can be reverted through an unlock settlement which is slower and incurs higher gas fees though guaranteed to succeed in a short time. As we presented in Appendix 4, the following sequence will be executed during the channel opening:
   - (1) *deposit* assumes that participants already transferred deposit to form the off-chain transaction.
   - (1.1) *checkStatus*, enum for object *Status* represents a group of constants with the following values: open, deposit, settle, destroy, closed and unlock transfer.
   - (1.2) *renew deposit*, if one of the opened channel participants has insufficient balance, then the state channel application will allow the participants to renew their deposits or reload their ethers into the token network.

   The initial state, after the publication, is successfully settled during the channel opening. Locking and unlocking the number of deposits from all involved parties are performed using an on-chain operation on the Ethereum network. This will lead to further off-chain transactions. Appendix 1 describes the process for the channel opening in detail.
2. **Settlement**. This state also refers to updating the state after the channel has been established and participants exchange assets in the root state. In this phase, the parties will transact by directly exchanging states between them using *OutgoingChannelState* and *BlockchainChannel*. We demonstrated the object channel node implementation in Appendix 2 and Appendix 3. The *OutgoingChannelState* transactions will modify the initial state and distribute the blocked assets among the participants. All parties must approve the agreement to a new state by signing it and sending it to the other participants. The state's order is maintained by a version counter and *transfer_id*. The final state is published by default until the expiry of the challenge period. Each participant can then withdraw the amount corresponding to them in the channel. In the settlement process, the following code with 1.2.1.1:λ will be executed, and the asynchronous transaction process will be directly communicated with the smart contract wrapper.
   The *OutgoingChannelState* will manage state transitions after channel opening. Subsequently, the channel status traverses from the following: (i) settled state, indicating that the deposit has been approved; (ii) closed state, indicating that the channel is destroyed; and (iii) request closed state, where the channel can be shut down. Once the participants are in the settlement state, all parties can do off-chain transactions.

We examine the execution in Fig. 5.15 with preconditions for the channel opening.

- (1) *registerTransfer*: We assume that the participants have sufficient assets for exchange and initiated to form off-chain transactions by registering the transfer.
- (1.3) *getChannelAddress*: The transaction will be registered as per channel address, where the parties subscribe to the channel topic.
- (1.3) *getValue*: The transfer amount is verified, and it checks the sufficiency of participants' funds.
- (1.4) *verifyTransfer*: The channel participants verify the transfer if the deposit amount is sufficient.
- (1.4.2) *getClientAddress*: If the transfer has been verified, then the channel node will check the recipient address.
- (1.5) *getTransferId*: If beneficiary address is valid, the transaction that has been executed will return the *transfer_id*.
- (1.6) *isLocked*: The node locks the transfer until the beneficiary claims the assets that have been transferred. The unlocked process is according to the block hash and signature key.
- (1.7) *getValueWei*: If the block hash and signature key are valid, then the beneficiary can transfer the token into their wallet.
- (1.8) *stateChanged*: Once the transfer is unlocked, the iteration process for the state change will take place. Thus, all the channel parties can have more off-chain transactions and invite new participants to join their channel.

After channel establishment, other participants can renew their deposits by transferring some assets into the *SCHToken* object as detailed in Appendix 5. The number of channel participants per channel address is set at 10 per connection.

3. **Closing**. After completing transactions and achieving a final state, parties may submit the final state to the ledger contract. The *ChannelContract* object validates the published state by matching the signatures with the initial state, settles the channel balances, and pays out to each participant.
   We examine the execution in for the channel closing and checking the settlement process.
   - (1) *isCloseRequested*: We assume that participants have sufficient channels to close, and the token asset has been settled into their online wallet.
   - (1.3) *getCloseRequested*: The node server will receive a message to close the channel, and the *ChannelContract* wrapper object will verify the address. If a block transaction is <= 0, the channel will be shut down gracefully.
   - (1.3) *getPendingStatus*: Any pending transaction in the channel that is being closed will be checked, and then the channel node will wait until that pending transaction is settled. The asset is transferred to the recipient's address.

   If other participants find the registered state identical to the final state, then no action is required from them. After the challenge duration expires, the

state will be finalized on-chain. If a participant discovers that the state is not the most recent, then he can refute it by submitting the most recent state. A sample state channel transition is shown in Rinkeby Test Network[18] that contains the open and close. For example, in the column **Before** is represented the opened state and the channel participants shown in the left column **address**. The close state is shown in the column **After** and the **State Difference** column indicate the block nonce already updated outside the test network. In this case, the settled transaction state is not broadcasted on the Rinkeby test network and affects the block size collected for each transaction settled via the blockchain network.
The implementations of the closing transaction according to the process above can be found in Appendix 4 and Appendix 7.

3 Experimental Set-up

In this research, channel node components were deployed on multiple virtual machines, running RHEL v8 x86 64 (development license) built on Google Cloud Platform (GCP). To fulfill the most minor hardware requirements, 4 virtual CPU cores, 16 GB memory, and 20 GB persistent storage space are provided to every VM. The Redis data store saved off-chain data-related transactions, and Apache Tomcat v8 served as an application server. Inter-region type deployment is performed in this study, where the channel node server datastore is deployed in the Asia-southeast2-a zone Europe-west3-c zone, respectively. According to GCP documentation[19], the implementation of the internal IP address is suggested over external IP in order to get maximum throughput.

The evaluation performed for off-chain transaction is represented by *SignedTransaction* object whereas on-chain operation is demonstrated by *SignedTransferUnlock*. According to [5], the maximum transactions per second (tps) are calculated using:

$$tps_{on-chain} = \frac{\mathrm{Gas}_{Limit}}{\mathrm{T}_{xGas} * \mathrm{Block}_{Time}} \tag{1}$$

where Gas_{Limit} is the block gas limit, Tx_{Gas} is the gas needed to compute the simplest transaction and $Block_{\mathrm{Time}}$ is the blockchain block time. In our channel node server, all the above tps parameters can be configured under application properties (*application.yaml*). The result for on-chain transaction throughput, which we have observed, is presented in Fig. 7. The tps time across 10 transactions was measured for each of the two processes. The results of these measurements are presented as a cumulative transaction in Fig. 7 which will significantly be decreased by pushing new inter- block times. Therefore, for each inter-block times request, the gas price was increased along with the block gas limit. As a result, the throughput for the on-chain transaction was 180 s for six on-chain processing, respectively, and inter-block times were appended after 60 s (we incrementally increased the transaction fees to avoid network timeouts). Contrary

[18] Sample state channel transition https://rinkeby.etherscan.io/tx/0x6824.
[19] Network throughput information https://cloud.google.com/compute/docs/network-bandwidth.

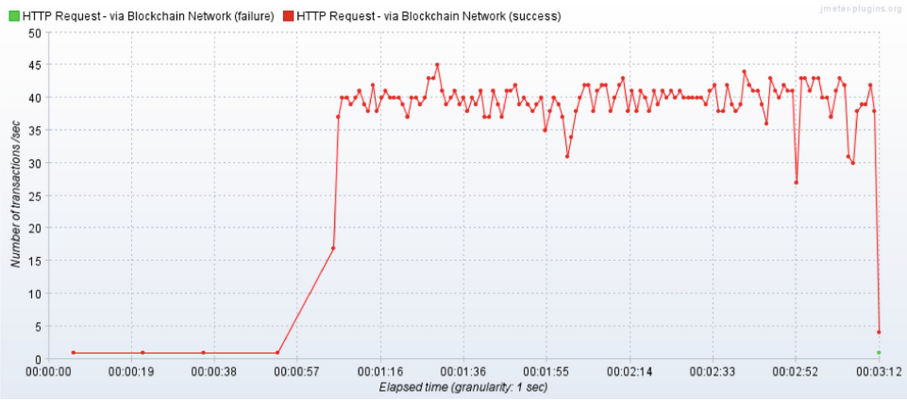

Fig. 7. On-chain transaction via Rinkeby network and gas price 99 GWEI

to on-chain transaction processing for the observed period throughput, the off-chain transaction can be achieved earlier than twice the median from on-chain computation, which is about 1.4 ms for 7 transactions. The observed result for off-chain transaction throughput is depicted in Fig. 8. In this context, transaction throughput refers to the number of requests processed by the server per time unit (seconds, minutes, or hours). The transaction processing times will be faster if the system generates a higher throughput and a more stable transaction success rate. In the off-chain computation that we simulated, we could achieve higher throughput by 1,395,877 in 120 s (around 96 tps). Additionally, all off-chain transfers were guaranteed to be received by other channel participants if the channel closed successfully (Fig. 7).

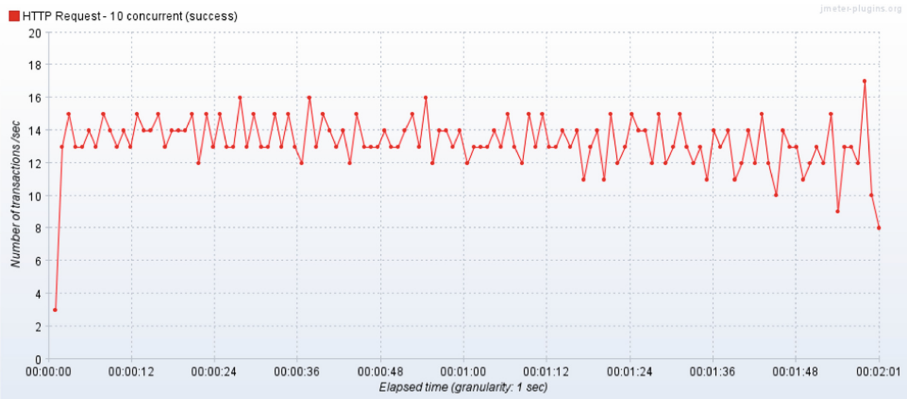

Fig. 8. Off-chain transaction throughput

The throughput report provided in this test is limited to serve the number of tps which can be achieved by off-chain computation. Hence the comprehensive benchmark report will be conducted in the next release. Furthermore, it is planned in the next test iteration to evaluate the overall system benchmark that will include the transaction inclusion time, inter-block time, and smart-contract execution time (in-particular *SCHToken* that implements the ERC-20 library).

4 Conclusion

L2 techniques have emerged as promising scalability solutions to improve the quality of blockchains services, mainly to increase transaction throughputs. The MLR methodology was adopted to design the research steps that guide the achievement of the research objectives and answer the related research questions. For example, the MLR process, awareness of problems and suggestions, was used to gain theoretical knowledge and understand the problem domain. The existing L2 scalability platforms were examined and compared to visualise their functionalities. Only a few of the L2 scalability proposals are currently viable. The main objective of this research is to review solutions that can help achieve scalability solutions in Ethereum blockchains. Based on this work, we planned to have RDF store technology in our solution, so a full-pledge feature from the RDF store engine was not yet implemented. The token network is the first functionality we can incorporate in our channel state application. In particular, we can allow channel participants to deposit their assets using the object and as long as there is another participant who will approve their token deposits. The second functionality is the channel-node application, a multi-user node server intended to facilitate users to open, transact on and settle state channels. Finally, our primary development goal was to demonstrate that our channel protocols, business processes, and underlying intelligent contracts work by the initial MLR. We work on the L2 techniques in the proposed research work to improve the scalability.

This extension of research area will provide instant payments without any intermediaries and the core engine that is backed up by nanopayments with the full-feature RDF store, as well as dimension of Machine-to-Machine Payment similar to μRaiden[20], and will analyze the impact of our proposed solution on the systems security and users privacy.

Appendix 1 Payment Channel Network

1.1 Go-Perun

A concept termed "virtual channels" was introduced by Go-Perun [13], which is used to resolve again the shortcoming mentioned previously. Here, for example, we assume that Bob and Alice are both linked through a blockchain channel

[20] https://raiden.network/micro.html.

provided by Ingrid, an intermediate payment hub. A direct linkage can be set up between Bob and Alice through a virtual channel using these ledger channels. Here, the involvement of intermediary Ingrid is not required in every payment. Owing to this, latency and costs are reduced, and privacy is protected. The process for the state channel, as detailed in [13], can be visualised in Fig. 9, which works on-chain only when the participants open and close the channel. Once the channel is established, the transaction will be moved outside the leading network, proceeding with an off-chain computation.

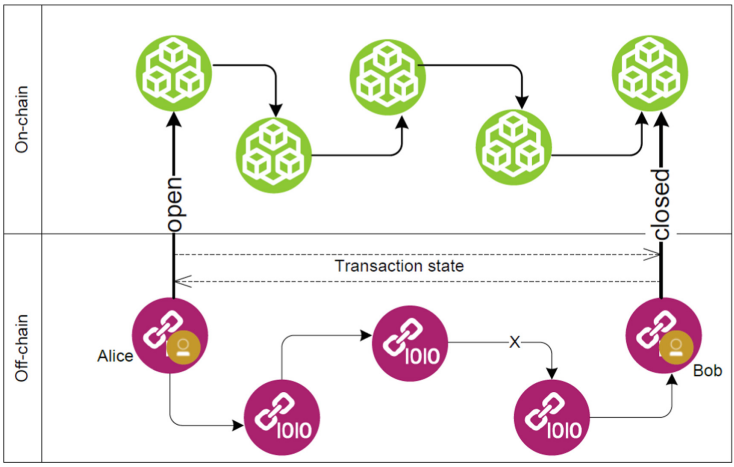

Fig. 9. On-chain and off-chain transactions.

The Go-Perun framework begins with an explanation of ledger payment channels. These channels are formed by interacting with the blockchain, where instant mutual payments are allowed between two parties. In an opening operation, a ledger payment channel β is formed between Bob and Alice, where xA coins and xB coins are deposited by Alice and Bob, respectively. As a result, the channel's balance is represented using [Alice 7! xA; Bob 7! xB]. It shows that Alice has xA and Bob has xB coins. Thus, the value of the channel is xA + xB. These coins will be kept blocked as long as the channel β is closed and cannot be used by the parties. Bob and Alice can modify the division of funds in the channel upon finishing the setup. Payments between Bob and Alice are made through the updated methodology.

Virtual channels are based on the concept of repeatedly applying the channel technique by creating a virtual payment channel, which is on top of the ledger channels. Assume two ledger channels among Bob, Alice, and Ingrid: βA between Alice and Ingrid and βB between Ingrid and Bob. In [13,14].

1.2 Raiden Network

The Ethereum community has approved the Raiden protocol as the first prominent off-chain computation for the L2 scalability solution. The Raiden Network has published all the sources in this section in [30] as grey literature. Hence, in this research, we have formalized the literature as white literature that academics have also cited for a mature solution implemented in L2 scalability inside the Ethereum network. An off-chain scaling solution, which enables token transfers that are quick, low cost, and scalable, is termed the Raiden network. It works with any token compliant with ERC-20 and supports the Ethereum blockchain. The Ethereum blockchain implements the balance proof for channel participants in the Raiden network. A Raiden balance proof binds like an on-chain transaction as only two involved parties can approach the tokens transferred to the smart contract of the payment channel. In this agreement, several others are shifted to the token contract as a pledge.

The Raiden network offers two core components: (i) when participants open the channel, the **token network** is used to deposit ethers, and (ii) with considerably enough space between the sender and receiver, the way of payment channels is **routing network**.

a. **Token Network**
To complete a token transfer, asset transfers employ numerous payment channels. They allow users to make transactions to other users with whom they do not have a routing network. Alice uses the channels among herself, Bob, Charlie, and Dave to transfer a payment to Dave, as shown in Fig. 10. A secret key is needed to claim this amount for tokens until it stays locked as a pending transfer. Dave asks Alice for the secret key after he receives the transfer. Dave receives the secret key from Alice and uses it for unlocking the pending transfer. This secret key is then sent to Charlie, and he signs a balance proof. Afterward, the secret key is sent back to Dave. Then, the process is reversed until everyone in the mediated transfer channel knows about the secret key.

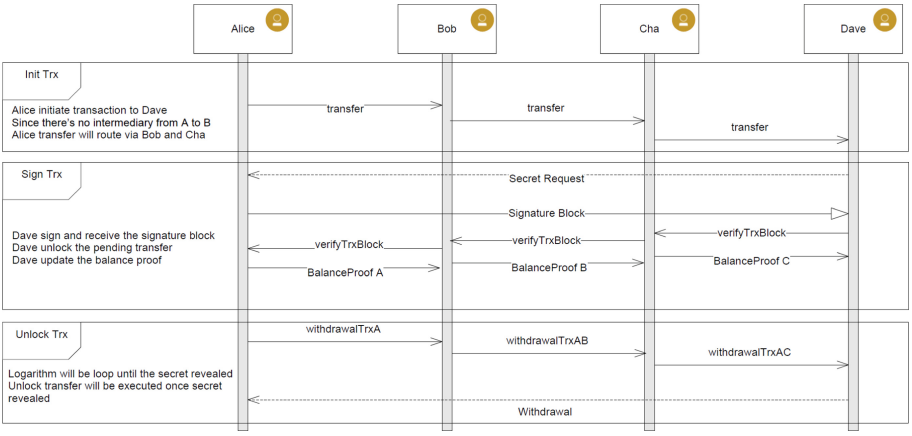

Fig. 10. Raiden token network transaction.

b. **Payment Routing**
A route of payment network with adequate capacity channels is required between the sender and receiver for a mediated transfer. Capacity refers to the sufficiency of tokens available in the payment channels for establishing pending transactions. In Fig. 11, we present payment routing in the following mechanisms:
- The capacity is adequate as Bob and Alice have five tokens.
- Payment is allowed to be continued because there are four tokens in the custody of Bob.
- This route is not viable as Charlie and Dave have two and eight tokens, respectively.
- A refund transfer is generated by Charlie, which Bob will claim.

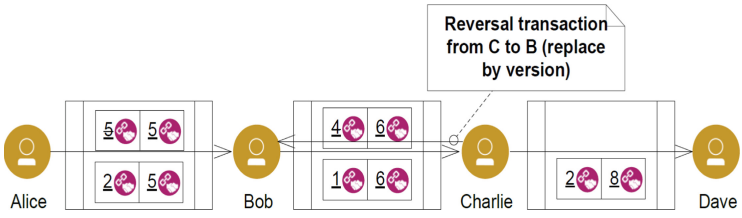

Fig. 11. Payment routing using the Raiden protocol.

A route of a linked network having a channel capacity of three in the mediating payment channel is required by Alice to pay three tokens to Dave. The complete view of the network is contained in the Raiden network protocol [31], and simplified flow is in Fig. 10. Assume that each user has deposited five tokens and performed several transactions. With the discovery of a path with sufficient capacity, the Raiden network will find a different path.

To make sure that the token liability will be paid by the involved participants in a smart contract, tokens must be locked up as a deposit for the complete lifecycle of the PCN. Until the final closing by any of both participants, this deposit guarantees that tokens may only be used to send to the channel partner and receive tokens from an adequate peer. This is how the double spending of tokens to other peers is prevented for both participants.

The involved parties are free to send certified checks back and forth after establishing a channel. However, every peer maintains a copy of the most recent check rather than monitoring all checks. The balance proof is a document signed by the sender digitally that details the final total of all Raiden transfers, which are delivered to a participant up to a specific point. A channel always keeps both of the participants involved in it, and they act as a bar tab of the channel. Various alternating credits are traded, altering the overall amount due between the channel members and perhaps re-balancing the payment route several times.

In our state channel application, we have a modified token network integrated into our system. The *TokenService* object will directly communicate with the

state channel token (*SCHToken*) that we have deployed inside the Ethereum network. Figure 12 shows how the *TokenService* works with *SCHToken*, and the following process will be executed during the state channel life cycle.

- (1) First, the actor will initiate token funding. This process will be taken care of by the *TokenService* apart from the payment channel object. The remote function call will go to the *SCHToken* inside the blockchain network, and it will take the responsibility and approve the amount of ether that will be saved inside the token network (1).
- (2, 3, 4) Next, the response message will be saved into the Redis data store via the modified RDF4LED engine [21].
- (5, 6) After saving the transaction record into the data store, off-chain computations (e.g., transfer, withdrawal, and reversal) will directly be controlled between the payment channel and Redis data store.

During the simultaneous opening of the channels, every network participant will have more than one transaction. Due to this reason, the renewal of the deposits in the payment channel is expected in every transaction. Hence, *TokenService* makes it comfortable to reload new deposits into the existing open channel over the network of channels.

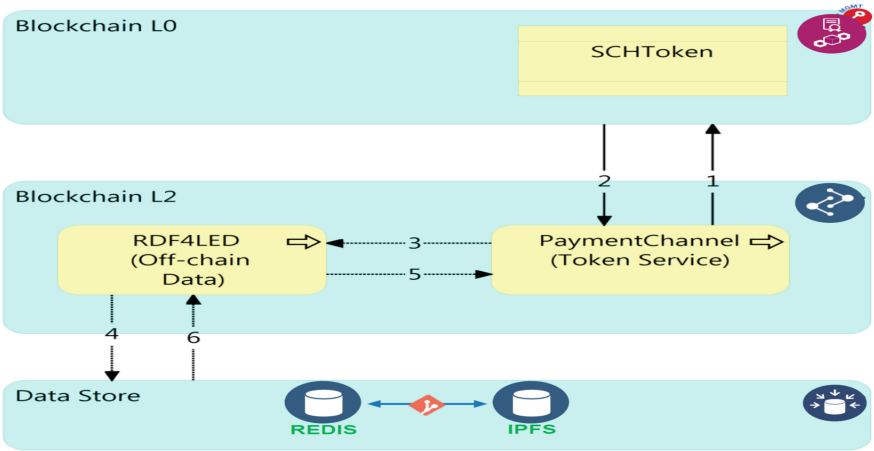

Fig. 12. Off-chain transaction service

Appendix 2 Watchtower Committee

Watchtower committee in Fig. 13 proceed as follow:

- (1) The *OutgoingChannelCoordinator* object will start the watchtower if the open state has been established,
- (1.1) If the channel already in open state, in this context the deposit has been approve by both participants, then

– (1.1.1) The *ChannelCoordinator* will assigned the watcher name to monitor the channel lifecycle,
– Each of watchtower committee object will be deputed to monitor specific channel process, hence, in the sequence flow it will return valid channel address properties.
– This monitoring service will continue until the channel is shutdown.

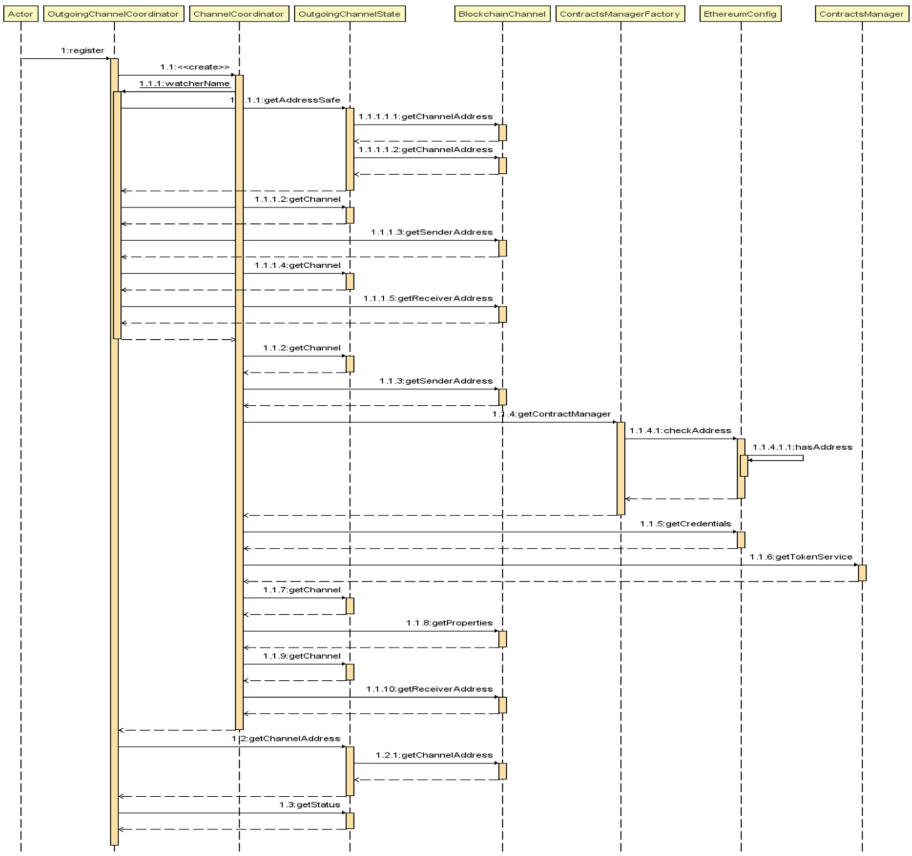

Fig. 13. Watchtower committee.

Appendix 3 Channel composability

One of a feature that we offer inside channel node application is to enable new participants join on the existing channel that has been established. This process in Fig. 14, refer as "Channel Composability" that will be execute as follow.

– (1) If there is new channel participant join the network, the *ChannelJoinImpl* object will start the process execution, by sending new member participant properties into the *ChannelPoolProperties*

- (1.1.1) property values will be check accordingly by *OutgoingChannelPolicy*, if the properties satisfied then in sequence (1.1.10) the new member will added in the existing *OutgoingChannelPoolManager*.
- (1.1.3) once the channel properties updated, the new network configuration will be add into *ChannelPoolProperties*. Those new configuration include the sub-sequence (1.3) *getSettleTimeOut* and (1.7) *getDeposit*.
- (1.10.2.2.λ) the process will loop if there is any new channel participant would like to join the existing network, in this case the looping process will break if the channel capacity is reach n $<$ 10. The node server allow to override existing channel capacity via *application.yaml*.

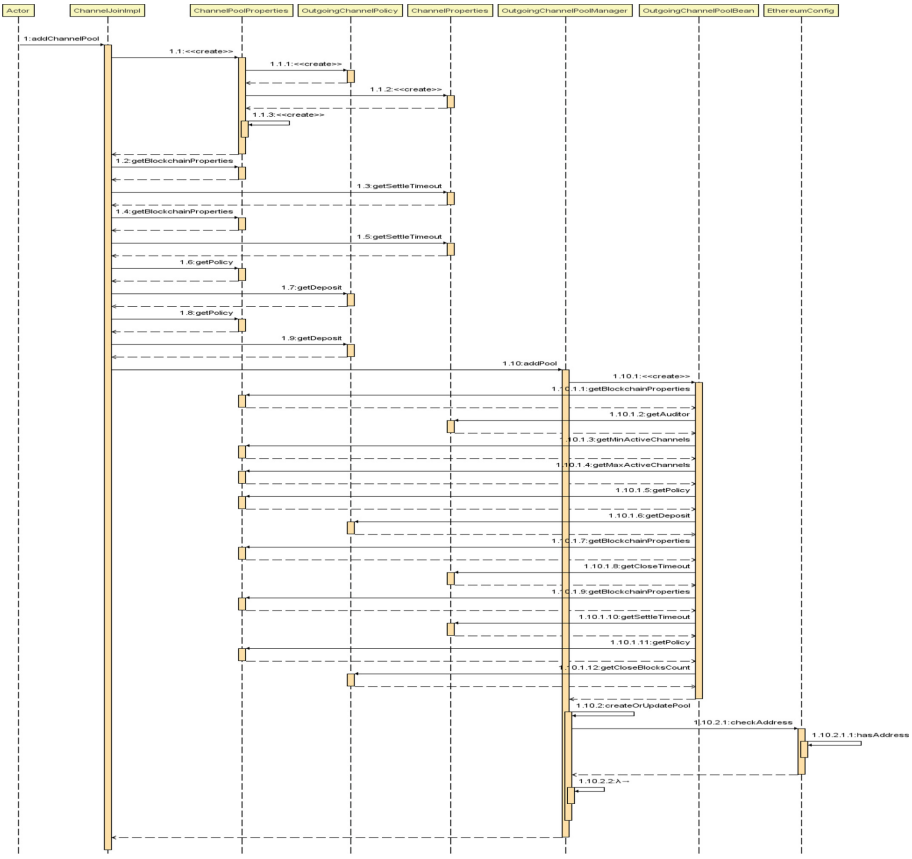

Fig. 14. Channel composability (join channel process).

Appendix 4 Close Channel Sequence Diagram

See Fig. 15.

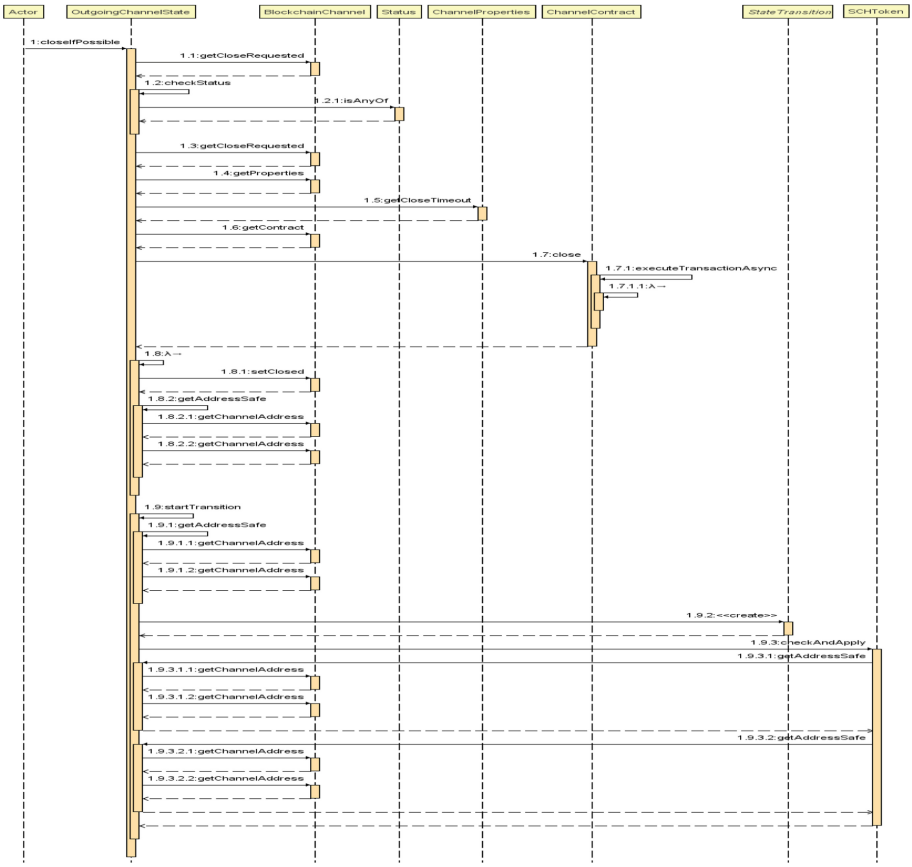

Fig. 15. Closed channel.

References

1. Abadi, D., Arden, O., Nawab, F., Shadmon, M.: Anylog: a grand unification of the internet of things. Cryptography (2018)
2. Adrian, L.: Gartner predicts 90 implementations will require replacement by 2021. https://www.gartner.com/en/newsroom/press-releases/2019-07-03-gartner-predicts-90-of-current-enterprise-blockchain
3. Breidenbach, L., et al.: Chainlink 2.0: next steps in the evolution of decentralized oracle networks (2021)
4. Buterin, V., Van de Sande, A.: Eip-55: mixed-case checksum address encodin (2016). https://eips.ethereum.org/EIPS/eip-55

5. Caccia, F.: On ethereum performance evaluation using PoA (2019). https://blog.coinfabrik.com/on-ethereum-performance-evaluation-using-poa/. Accessed 21 July 2021
6. Cannell, J.S., et al.: Orchid: a decentralized network routing market (2019)
7. del Castillo, M.: The 10 largest companies in the world are now exploring blockchain. https://www.forbes.com/sites/michaeldelcastillo/2018/06/06/the-10-largest-companies-exploring-blockchain/#fc1e3f61343d
8. Coleman, J., Horne, L., Xuanji, L.: Counterfactual: generalized state channels (2018). Accessed 4 Nov 2019
9. Connext: Crosschain liquidity network for ethereum. https://docs.connext.network/. Accessed 10 Apr 2020
10. Crain, T., Natoli, C., Gramoli, V.: Evaluating the red belly blockchain. arXiv preprint arXiv:1812.11747 (2018)
11. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Pelc, A., Schwarzmann, A.A. (eds.) SSS 2015. LNCS, vol. 9212, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21741-3_1
12. Dong, M., Liang, Q., Li, X., Liu, J.: Celer network: bring internet scale to every blockchain. arXiv preprint arXiv:1810.00037 (2018)
13. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: virtual payment hubs over cryptocurrencies. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 106–123. IEEE (2019)
14. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 949–966 (2018)
15. Galal, H., Elsheikh, M., Youssef, A.: An efficient micropayment channel on ethereum, pp. 211–218 (2019)
16. Grubenmann, T., Bernstein, A., Moor, D., Seuken, S.: Financing the web of data with delayed-answer auctions. In: Proceedings of the 2018 World Wide Web Conference, pp. 1033–1042 (2018)
17. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: SoK off the chain transactions. IACR Cryptology ePrint Archive **2019**, 360 (2019)
18. Hafid, A., Hafid, A.S., Samih, M.: Scaling blockchains: a comprehensive survey. IEEE Access **8**, 125244–125262 (2020)
19. KChannel: Kchannels is a new payment channel platform for ethereum. https://docs.kchannels.io/docs. Accessed 10 Apr 2020
20. Khalil, R., Zamyatin, A., Felley, G., Moreno-Sanchez, P., Gervais, A.: Commit-chains: secure, scalable off-chain payments. Cryptology ePrint Archive, p. 642 (2018)
21. Le-Tuan, A., Hingu, D., Hauswirth, M., Le-Phuoc, D.: Incorporating blockchain into RDF store at the lightweight edge devices. In: Acosta, M., Cudré-Mauroux, P., Maleshkova, M., Pellegrini, T., Sack, H., Sure-Vetter, Y. (eds.) SEMANTiCS 2019. LNCS, vol. 11702, pp. 369–375. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33220-4_27
22. McCorry, P., Bakshi, S., Bentov, I., Meiklejohn, S., Miller, A.: Pisa: arbitration outsourcing for state channels. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies, pp. 16–30 (2019)
23. Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: payment networks that go faster than lightning. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 508–526. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_30

24. Network, P.: Blockchain thunder. https://github.com/blockchain/thunder/. Accessed 10 Aug 2020
25. Network, P.: Papyrus network. https://github.com/papyrusglobal/papyrus/. Accessed 10 Aug 2020
26. Panetta, K.: Trends emerge in the gartner hype cycle for emerging technologies (2018). Accessed 5, 6 June 2019
27. Paul, B., Erik, M.: Eip-1474: remote procedure call specification [draft] (2018). https://eips.ethereum.org/EIPS/eip-1474
28. Poon, J., Dryja, T.: The bitcoin lightning network: scalable off-chain instant payments (2016)
29. R3: R3 corda transactions per second (TPS). https://www.corda.net/blog/transactions-per-second-tps/
30. Raiden: Off-chain scaling solution, enabling near-instant, low-fee and scalable payments. https://raiden-network.readthedocs.io/. Accessed 10 Apr 2020
31. Swan, M.: Blockchain: Blueprint for a new economy, vol. 26. O'Reilly Media, Inc. (2018). Published 24 Jan 2015
32. Web3Labs: Develop on ethereum with the JVM. http://docs.web3j.io/latest/quickstart//. Accessed 10 Aug 2020
33. Xu, X., Weber, I., Staples, M.: Architecture for Blockchain Applications. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-03035-3