A Middleware for Hybrid Blockchain Applications: Towards Fast, Affordable, and Accountable Integration

Yessenbayev, Olzhas; Comuzzi, Marco; Meroni, Giovanni; Nguyen, Dung Chi Duy

Published in:
Proceedings of 21st International Conference on Service-Oriented Computing

Link to article, DOI:
10.1007/978-3-031-48421-6_21

Publication date:
2023

Document Version
Peer reviewed version

A Middleware for Hybrid Blockchain Applications: Towards Fast, Affordable, and Accountable Integration

Olzhas Yessenbayev[1][0000−0002−5830−4715], Marco Comuzzi[1][0000−0002−6944−4705], Giovanni Meroni[2][0000−0002−9551−1860], and Dung Chi Duy Nguyen[1]

[1] Ulsan National Institute of Science and Technology, Ulsan, Korea
{yess,mcomuzzi,int2k}@unist.ac.kr
[2] Technical University of Denmark, Kgs. Lyngby, Denmark
giom@dtu.dk

Abstract. Hybrid blockchain architectures combine centralized applications, like enterprise systems, with (public) blockchain to implement additional functionality, such as tamper-proof record keeping. To reduce the latency and cost of using a public blockchain, these systems may rely on batching of transactions or general-state channel networks. While reducing costs, the former increase the latency. With the latter, only major state updates are recorded on-chain, while most transactions history remains only on the channels. This paper describes a novel solution that combines the benefits of both approaches to decrease the latency and cost of hybrid blockchain applications. We propose to combine a local blockchain that runs on a centralized server to provide near-immediate state update confirmation, with a batching mechanism sending transactions to a public blockchain for record-keeping at a most convenient time. We also introduce a dispute mechanism promoting the prompt delivery of correct batches to the public blockchain by the application provider, thereby deterring malicious behaviours. The solution is motivated by a fintech use case, for which we also show the implementation of a prototype and an experimental evaluation of the latency and cost savings.

Keywords: Blockchain, Ethereum, gas price, latency, meta-transaction delegation

1 Introduction

Blockchains, as transparent and open databases, enable immutable records traceable to the original signer; through smart contracts, they can provide automated rule enforcement [1]. Originally developed for decentralized peer-to-peer applications, blockchains also enable the so-called *hybrid blockchain architectures*, where users interact with traditional centralized applications augmented with blockchain [2]. Specifically, the use of public blockchains presents intriguing scenarios in enterprise applications. For instance, a corporate enterprise resource planning (ERP) system paired with the Ethereum blockchain can provide

Pre-print copy of the manuscript published by publisher="Springer Nature Switzerland", (available at link.springer.com)

tamper-proof record keeping and digitized asset tracing. Platforms like Provenance leverage this to prove product authenticity on public blockchains [3], while MedRec employs the Ethereum blockchain to offer secure medical records, thus enhancing healthcare data interoperability [4].

The architectural limitations of public blockchains, such as latency from a few seconds to several hours [5] and volatile transaction costs depending on network congestion and transaction complexity [6], still limit the enterprise-wide adoption of hybrid architectures. To address these issues, batching and general state channel networks have emerged as viable solutions. Batching involves sending multiple transactions at once, instead of individually. In this way, the fixed per-transaction costs [7] can be reduced. However, latency could increase, since the transactions are not processed until the whole batch is sent. General-state channel networks allow near-instant interactions between participants by simulating smart contract state transitions [8] until the final state is posted on the public blockchain. However, most of the transaction history is recorded off-chain. Therefore, this option may not suit hybrid blockchain applications, where the transaction history often serves as a comprehensive, tamper-proof record for traceability and auditing purposes.

In this paper, we present a novel solution that combines the strengths of the two approaches described above. We propose a middleware system that can be integrated with a centralized application to provide reliable and immutable record-keeping on a public blockchain, limiting the application latency and costs. The system consists of two key components: (i) a local private blockchain to instantly simulate state transitions and (ii) an asynchronous process batch-transferring local transactions to the public blockchain network. A *target* smart contract used for record-keeping on the public blockchain is replicated on a local private blockchain, enabling near-instantaneous transaction confirmation. The transactions are then cost-efficiently batch-sent to a public blockchain network for permanent and immutable record-keeping. By maintaining the transaction order within batches, the proposed approach guarantees consistent execution across both networks. We also introduce a *dispute* mechanism that creates an incentive for the application provider to eventually send all the correct batches to the public blockchain, thus preventing opportunistic or generally malicious behaviours. The proposed approach is inspired by the needs of a real-life fintech company in South Korea providing a flexible salary payment service that uses a public blockchain for transparent record-keeping. Besides describing this use case, we also experimentally evaluate the proposed approach in terms of cost reduction as a function of the batch size, and latency improvement.

The remainder of the paper is organized as follows. Section 2 introduces the use case and the requirements, while Section 3 presents our solution. Section 4 discusses the implementation and the experimental evaluation. Section 5 compares our solution with existing literature. Finally, Section 6 draws the conclusions and outlines future work.

2 Problem definition

Section 2.1 introduces the fintech use case that inspired the development of the proposed solution, whereas Section 2.2 extrapolates more general requirements that the proposed solution must address.

2.1 A Fintech Motivating Case Study

GivingDays Inc. [3] is a financial intermediary providing a flexible payment service (Thankspay) enabling employees of partner companies to request salary advances through their application. The advances are immediately paid by GivingDays (that is, earlier than the scheduled pay-day) and charged to the partner company later.

While managing funds purely in cryptocurrency is infeasibile, the ThanksPay service can leverage public blockchain to ensure traceability, transparency, and asset reusability. Through private key encryption, blockchain can allow to publicly verify that each salary advance request genuinely originates from a worker, and is not forged arbitrarily by the ThanksPay service to maliciously charge partners. This eliminates the need for laborious server audits. Smart contracts also enable automated and transparent compliance with salary withdrawal regulations. Finally, the open nature of public blockchains allows to extend the service to other use-cases, such as flexible loans.

2.2 Requirements

Inspired by the Thankspay use case, we identified a set of requirements that must be addressed by a more general system that provides transparent record-keeping by adopting a hybrid blockchain architecture.

R1: Minimize application latency: The latency of public blockchain is usually high and unpredictable, which might jeopardize an application whose logic relies on blockchain as a database. For instance, the Ethereum latency is 15-20 seconds for simple cases and significantly higher for complex transactions.

R2: Minimize transaction fees: As the number of transactions to be recorded grows, the costs of facilitating them also increase. Moreover, transaction fees on public blockchains like Ethereum might be extremely volatile, posing additional challenges for a record-keeping system.

R3: Maintain full transaction history: Record-keeping systems must demonstrate an unalterable record of every transaction. This requires each transaction to go through rules-checking and storage on the public blockchain. Therefore, posting only periodic hash updates would not be sufficient.

R4: Guarantee transaction order and inclusion: Reordering or excluding user transactions can alter the state of the blockchain. If the correct ordering is not enforced, the service owner can exploit this for its own benefit.

[3] `https://www.thankspay.co.kr/`

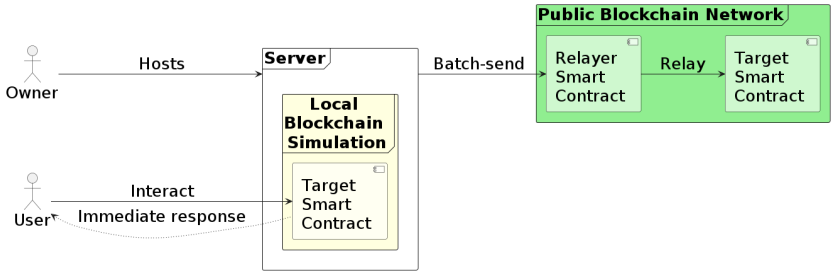

Fig. 1: Overview of the system

R5: Provide a balanced approach to user self-custody: It would be infeasible to expect each user to own a cryptowallet to fund and manage transactions. On the other hand, centralizing the entire process within a backend service recording all the transactions on behalf of the users would nullify the trust brought by blockchain technology [2].

3 Solution design

We propose a middleware solution functioning as an intermediary between users and the public blockchain. The static system components are outlined in Section 3.1, with a use-case scenario discussed in Section 3.2. Economic incentives ensuring the service owner's adherence to the batch-process (i.e. the dispute mechanism) are explained in Section 3.3.

3.1 System architecture

A high-level overview of the proposed middleware solution is sketched in Fig. 1. It involves the following entities:

- **Owner**: the host of the service (e.g., GivingDays hosting the ThanksPay service in our motivating use case), which deploys smart contracts on the local and the public blockchains.
- **Users**: the end-users of the owner's service (e.g., the workers and partner companies in the motivating use case).

User actions are recorded as invocations of a *Target smart contract* deployed at a *public blockchain network*, representing the specific business application logic (i.e., logic of the Thankspay service in the motivating use case, determining, for instance, when workers can receive their salary in advance). To facilitate the user interactions with it, the owner instantiates a *local private blockchain* and deploys in it a replica of the target smart contract for the lifetime of the application. The local private blockchan instantly processes user invocations, enabling any business logic dependencies on the target smart contract state to be immediately fulfilled. Besides hosting the target smart contract, the public blockchain

Pre-print copy of the manuscript published by publisher="Springer Nature Switzerland",
(available at link.springer.com)

network also hosts a *Relayer smart contract*, responsible for batch-transferring transaction invocations from the local blockchain to the public blockchain's Target smart contract.

Users sign what are known as meta-transactions[2], transactions signed by a user's private key but sent by a third-party service provider. This technique allows users to interact with smart contracts without having to interact with the Ethereum blockchain, addressing **R5**.

The meta-transactions are immediately executed on the local blockchain for instant feedback (i.e., satisfying **R1**). When sufficient local transactions are accumulated, the server batch-transfers them to the public network, combining reduced per-transaction invocation costs with the possibility of strategically selecting the best time to send a batch to minimize transaction costs (addressing **R2** and **R3**). The users' signed messages contain a batch identifier and a relative position within the batch, guaranteeing the same order of transaction execution. By leaving a hash-trace of which transactions were executed inside of a public blockchain, we enable users to open and win monetary disputes against transaction exclusion from malicious owners, thus addressing **R4**.

To ensure state consistency between the local and the public blockchains, the *Target smart contract* follows constraints similar to the ones of general state channel contracts [9]:

1. **Limited access**: they should only be invoked by authorized users. In the local blockchain, this is ensured by giving access only to the users of the service. On the public blockchain, all calls to the target smart contract are routed through the Relayer contract.
2. **Insulation from external contracts**: they should not depend on external contract states, allowing the application state to be accurately predicted in advance, before on-chain execution.
3. **No global clock dependencies**: they should ban references to `block.timestamp`, given the unpredictability of when a transaction will be executed on the public blockchain.
4. **Modifying references to the senders of transactions**: in a meta-transaction executed via a relayer contract, `msg.sender` refers to the initiating address (i.e., the Relayer), not the original user. Therefore, the sender's identity needs to be included in the relayed transactions for the target smart contract to identify the user actually sending a transaction.

The latter can be addressed by passing the deciphered user address to the *Target smart contract* when invoked by the Relayer smart contract. This is done in an implementation-specific way (we discuss how we address this in Ethereum in Section 4).

3.2 System in-use view

The system usage involves two stages: i) local execution of meta-transactions, which users sign and exchange with the owner and ii) an asynchronous process batch-transferring these to the public blockchain.

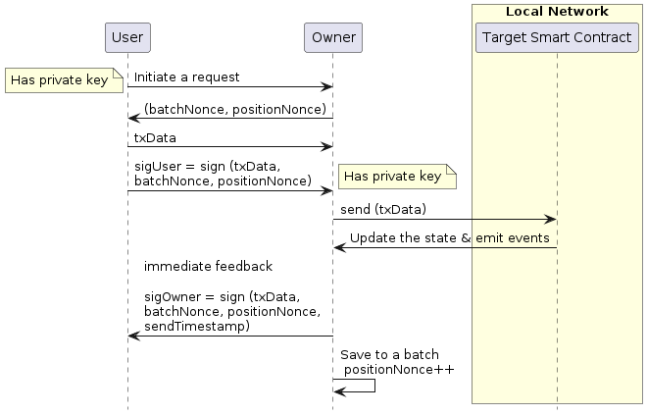

Fig. 2: Message exchange

Local execution of meta-transactions. Upon user registration, a unique private key is created locally within a user's applications. This private key can be used to produce a unique digital signature against a given message, ensuring that the signed data has not been tampered with (integrity of the message) and that the true signer's public account can be recovered (identity of the user). Users employ the private key to sign the digest (hash) of meta-transactions. In particular, the digest is computed from the following data:

(a) `txData`: a byte-encoded representation of the selected blockchain function (e.g., `requestSalaryAdvance`) and specified parameters (e.g., `amount`).
(b) `batchNonce`: a per-batch nonce to prevent malicious replay attacks.
(c) `positionNonce`: a transaction's position in the batch to prevent reordering and maintain execution order integrity.

To exchange these meta-transactions with the owner and obtain the owner's commitment to including the transaction into a batch, users follow the process outlined in Fig. 2:

1. Users *initiate a request* to the owner, who responds with `(batchNonce, positionNonce)`.
2. Users generate `txData` and their signature, `sigUser`, and send these to the owner.
3. The owner then *sends* the txData to the local network, which simulates the transaction, *updates the state*, and emits the necessary events for *immediate feedback* to the users. The owner also sends their signature (`sigOwner`) over the same parameters (with the addition of sendTimestamp, the timestamp by which the transaction is meant to be sent).

The `sigOwner` acts as proof of the owner's commitment to include the transaction into the next batch destined for the public network (important for the

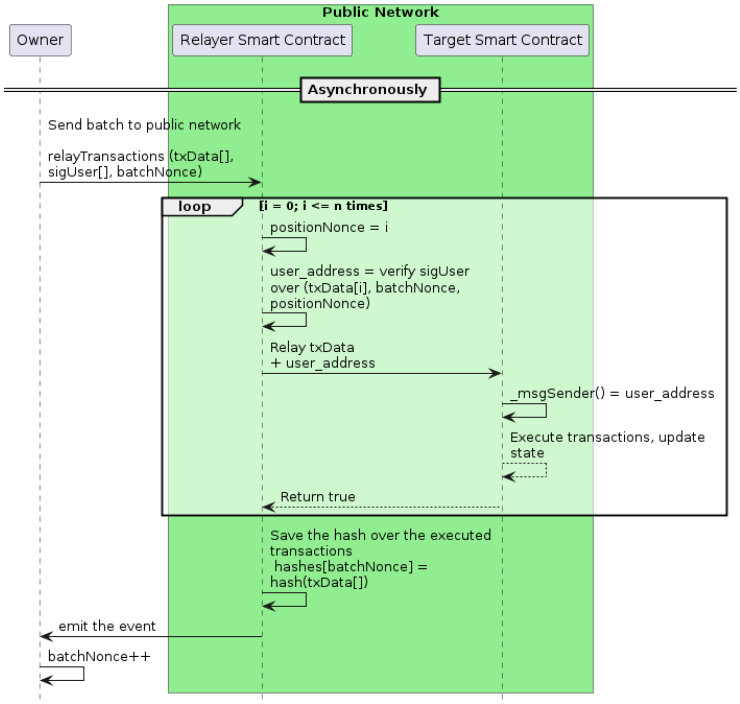

Fig. 3: Asynchronous sending

dispute procedure in Section 3.3). Until then, the users should assume the transaction has not yet been processed.

The owner can only process new transactions after the current one completes or times out. If a user fails to provide the necessary `txData` and `sigUser` within a specified timeout, the transaction is rejected and its `positionNonce` can be reused for the next transaction.

Asynchronous batch-transfer process. Once enough transactions are collected, the owner (see Fig. 3) *sends a batch to public network* through the Relayer smart contract. The `relayTransactions` function iterates through the meta-transactions. For each set of (`positionNonce, batchNonce, txData[i]`), the `verify` function decodes a user's public address from the message hash composed of these parameters and the given `sigUser[i]`. Any alteration to the message hash or signature will result in a different public address than the original one, preserving the integrity of the signed message and authenticating the signer's identity. The Relayer smart contract then relays `txData` to the target smart contract, together with the original user's public address. The target smart contract *executes the transaction* and *updates its state* (and corresponding digital assets) accordingly, on the behalf of the original user.

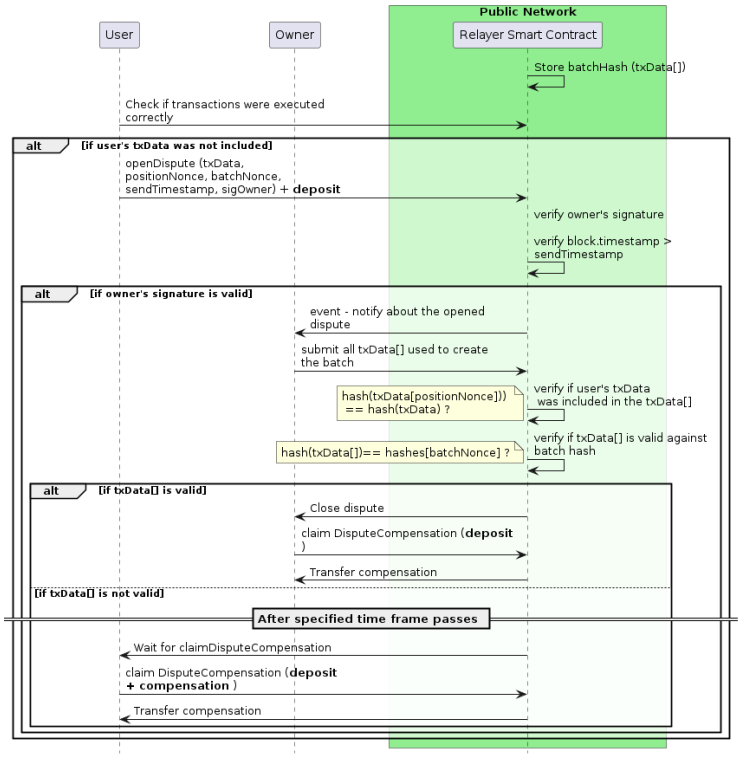

Fig. 4: Dispute procedure

Upon the completion of all transactions in the batch, the Relayer calculates and *saves the hash* over the executed transactions (`txData[]`), providing a reference for maintaining a consistent transaction execution order across batches. Afterwards, the smart contract *emits the event* signifying successful processing; the owner increments the nonce associated with the batch (`batchNonce`) off-chain, allowing for the processing of the next batch of transactions.

The signature verification coupled with user address relaying to the receiving target smart contract enables users to maintain unique and persistent on-chain identities in meta-transactions. The finalization of transaction batches and the storage of their associated hashes provide a traceable reference for maintaining accountability in the system, as detailed in the next section.

3.3 Maintaining accountability of meta-transactions

A malicious owner can potentially disrupt the system by (i) not including certain local transactions in a batch sent to the public blockchain or (ii) modifying the order of execution of the transactions. In the Thankspay service scenario,

this may be driven by financial objectives to manipulate the workers' account balances and forge undue settlement.

To address this issue, we introduce a dispute procedure fully managed by the public Relayer smart contract that allows for the monetary punishment of any owner misconduct (see Fig. 4)q. This mechanism requires the users to make a deposit of amount of cryptocurrency `X` to open a dispute, discouraging frivolous claims. If the owner does not address a dispute within a specified timeframe, the user can claim a compensation of `X + Y`, disincentivizing the owners from any wrongdoing. To enforce compensation payouts for successful disputes, the Relayer smart contract is required to have sufficient funding to continue operating.

Implementing the dispute mechanism requires the following:

1. *Proof of the owner's commitment to include the transaction.* The owner is required to sign a user's transactions, with the addition of a `sendTimestamp`, as part of the feedback to the user (see Section 3.2). This signature, as `sigOwner`, ensures that transactions are committed for inclusion in the batch. Conversely, without this signature, the transactions are not "confirmed" from a users' perspective. If a transaction is signed by the owner and not sent to the public blockchain, the user can use these signatures as proof of commitment on the public blockchain that has not been fulfilled.
2. *Trace of transaction execution.* To leave an efficient trace of executed transactions in `relayTransactions` without storing the entire array, all `txData[]` in a batch are hashed together and stored in a mapping (`batchId => batchHash`).

The `sendTimestamp` is added to determine if the owner's promised timestamp has passed by comparing it to `block.timestamp`, therefore preventing premature disputes.

If the user is satisfied with the transaction execution, the process is concluded. If not, the user can initiate the dispute, which unfolds as follows (3.3):

(a) The user opens the dispute by invoking `openDispute` with the owner's signature as proof of commitment and the specified parameters (`txData + nonceId + batchId + sendTimestamp`). If *the owner's signature is valid* for these parameters, and `sendTimestamp > block.timestamp`, the dispute is successfully opened, and a corresponding event *notifies about the opened dispute* is emitted.
(b) The owner may close the dispute if they can provide evidence that the user's transaction was included and executed correctly. The owner *submits all `txData[]` used to create the batch* at a given `batchNonce` and calculates its hash. They win the dispute if the `txData` at `nonceId` matches the user's data, **and** the hash of the array `txData[]` corresponds to the recorded `batchHash` for that `batchNonce`.
(c) If the owner fails to resolve the dispute *after the specified timeframe passes*, the user is eligible to claim compensation.

Overall, this process safeguards against fraud. `sigOwner` only works for the given `(txData, positionNonce, batchNonce, sendTimestamp)` combination,

deterring users from fabricating commitments. The hashing of all `txData[]` on-chain also prevents false inclusion of a user's `txData` in the owner's proof, if it was not genuinely executed.

4 Implementation and Evaluation

The prototype implementation [4] of the Thankspay service is detailed in Section 4.1, with an experimental evaluation presented in Section 4.2.

4.1 Thankspay service implementation

Target smart contract. The Thankspay target smart contract is implemented as an Ethereum ERC-20 token, called *ThanksPaySalaryToken*, customized to manage salary advances, debt tracking, and settlements. The salary advances are implemented through "minting" and "burning" of the tokens. On a designated salary day, partner companies mint new tokens equivalent to the workers' salaries, replenishing balances and offsetting previous advances. As workers request salary advances, these tokens are burned, reducing the worker's balance and increasing the debt of the partner company, as tracked by the `partnerDebt` mapping. The debt is settled off-chain by transferring real funds to Thankspay, which is then reflected on-chain via the `settlePartnerDebt` function. The ERC-20 standard provides pre-defined functions and events for token management, making the solution more generalizable to other use-cases; token-burning functionality is inherited from `ERC20Burnable`. To minimize on-chain data, worker salaries are stored off-chain and passed as arrays of integers when required.

To allow the Relayer to pass the user address to the Target smart contract, we attach the user's address (authenticated from `msgHash` and `sigUser`) to the end of the relayed call data as `contractAddr.call(abi.encodePacked(txArray[i], msgSender))`. Secondly, within the target smart contract, we substitute all references to `msg.sender` with `_msgSender()` from the ERC2771Context [10]. This function interprets the tail-end of the call data as the sender's address, identifying the original user.

Relayer smart contract. We set both the deposit (X) and compensation (Y) values for `openDispute` to 0.1 ETH, with a dispute resolution time of one day. Disputes are managed within a mapping, `userAddress => disputes`. To ensure sufficient funds for dispute resolution, the smart contract is required to maintain a minimum balance of 0.5 ETH to continue operating (enough to cover five disputes), enforced by adding a modifier `onlyIfFunded` to the `relayTransactions` function.

Initially, the owner deploys the contract with 0.5 ETH and can add more funds through the `fund` function. User deposits also contribute to the contract's balance and could potentially be used to resolve earlier disputes. This approach

[4] Available at: `https://github.com/olzh-yess/A-Middleware-for-Hybrid-Blockchain-Applications`.

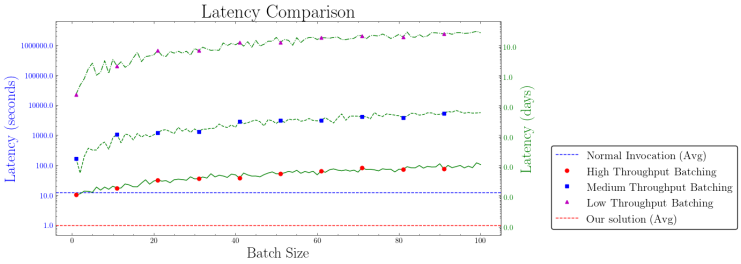

Fig. 5: Latency comparison

is justified, since successful dispute resolution results in a net loss of funds (0.2 ETH), implying that the owner would need to replenish the contract's balance to maintain the required minimum. This method is cost-effective as checking the smart contract balance consumes only 38 gas units, significantly less than the 2000 gas units needed to allocate and read a dedicated variable for tracking the owner's funds.

4.2 Experimental evaluation

We implemented a server prototype using NestJS with a WebSocket connection for real-time client-server message exchange. A persistent Ganache simulation and the Sepolia test network served as local and public blockchains, respectively. Simulated transactions are stored in SQLite database for easy batch-transferring.

We set up a typical workflow (deploying the target contract, enrolling partner companies, enrolling workers, processing salary advances, increasing chargeable balances, and resetting withdrawable balances on salary day) on the Thankspay service prototype. Then, we evaluated (i) the transaction confirmation latency and (ii) the cost (gas) savings. We compare our solution with two baselines: one in which transactions are sent to the public blockchain as soon as generated by the users, and one in which standard batching of transactions is used (but without local blockchain simulation to speed-up the confirmation). For the standard batching, we consider batch sizes from 1 to 100.

Latency savings. We define confirmation latency for the three considered scenarios as follows:

- **Public blockchain without batching**: Transactions are submitted directly to the Sepolia test network. To account for unpredictable network latency, we report average values from 10 tests for each smart contract function invocation, spaced at one-hour intervals.
- **Public blockchain with standard batching**: Transactions are reflected in the smart contract states only after an entire batch is completed and submitted to the public network. Consequently, latency comprises the public network latency and the rate at which new transactions are generated. For the latter,

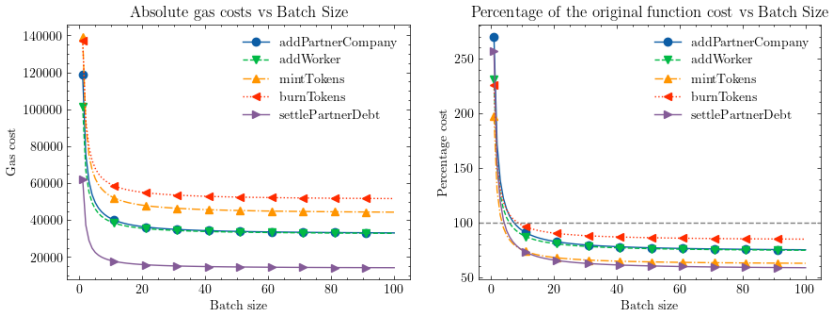

Fig. 6: Gas costs

we consider three hypothetical throughput conditions: (a) *high* - an average of 100,000 transactions per day (approximately 70 per minute), similar to Uniswap [11]; (b) *medium* - an average of 1 transaction per minute; (c) *low* - a scenario inspired by the ThanksPay service, with 3.35 transactions per day based on a scenario with five partner companies with an average of 100 employees, 20% of whom request their salary ahead of time each month.

– **Our proposed approach**: latency consists of the time required for users to sign and settle the results on a local network, as well as to get the owner's signature. We evaluated the average confirmation latency using the same settings as the solution described above without batching. The confirmation times on the public blockchain are expected to be similar to those associated with standard batching, as they are contingent on the time taken to generate a full batch.

Fig. 5 shows the results obtained. Our solution (0.1 s) is on average significantly faster than the baseline without batching (11 s). Note that the real Ethereum network may have latency up to four times higher than the Sepolia network [12]. The confirmation latency on the batching solutions is several orders of magnitude worse, especially for medium and low throughput scenarios.

Gas cost savings. We examine (see Fig. 6) gas costs of individual function invocations for batch transfers of varying sizes. Batching can notably reduce per-transaction *invocation* costs (e.g. miner fees and transaction verification), while not affecting the smart contract *execution* costs. Consequently, less complex functions, where invocation costs form a greater share of the total expense, benefit more from batching.

We can observe that gas reductions ranging from 15% to 45% can be achieved by setting batch sizes between 40 and 50, depending on the specific function. Increasing batch size further might be impractical, as the gas price reduction flatlines. Different timing policies of the batch-transfer can even further increase its efficiency (extensively explored in [7]).

Dispute procedure costs As disputes are expected to be opened infrequently, our primary focus is minimizing the amount of smart contract state stored for dispute resolution within the `relayTransactions` function. The only state-altering code instruction for disputes is saving the hash-trace of transaction execution for a given `batchNonce`,which adds a fixed per-batch additional cost of 30K gas units. In a batch of 40 transactions, this equates to a mere 750 gas units per transaction, a relatively low cost considering that the costs per transaction range from 20K to 100K gas units. Since it does not modify smart contract state, checking if the smart contract balance is higher than 0.5 ETH before the function invocation only adds negligible 38 gas units.

The remaining costs related to disputes are opening a dispute (149,623 gas units), closing a dispute (71,473) and claiming compensation costs (49,688).

5 Related work

The blockchain's architectural scalability can be achieved by altering its fundamental architecture (Level-1 or L1), or by introducing cheaper side-chains anchored to the native one (Level-2 or L2).

One approach to L1 scaling involves increasing transaction count per block or accelerating block generation frequency. This, however, faces the "blockchain trilemma" [13] - a trade-off between security, scalability, and decentralization. For instance, reducing block time could undermine consensus mechanism if new blocks cannot promptly reach all nodes. However, increasing the block size might preclude less powerful nodes from processing new blocks, threatening decentralization. This is illustrated by Binance Smart Chain's lower fees, but reliance on only 21 validator nodes [14].

L2 sidechains address the blockchain trilemma by processing a significant portion of transactions on smaller, faster blockchains with lower fees, while interacting with the main L1 chain when needed [15]. L2 sidechains ensure that the data is reliable, consistent, and unaltered by posting periodic updates, state hashes, or cryptographic proofs on the L1 chain. Although L2 sidechains inherit L1-level security for data integrity, they are still vulnerable to data availability attacks due to the smaller number of nodes and the off-chain nature of L2 transactions.

Batching services aim to reduce L1 gas fees by packing multiple meta-transactions within a single invocation, reducing the fixed per-transaction overhead costs. Meta-transactions encapsulate a user's desired action, the target smart contract address, along with unique signatures verifying the senders' identity; they are then batch-sent to the public network on the users' behalf by a central relayer and executed by a specially deployed dispatcher smart contract after appropriate verification [16] . Different batching policies have been explored in iBatch [7], while MultiCall [17] has explored hash-based authentication to decrease the costs of verification. The EIP-4337 [18] is a successful proposal to enable meta-transactions in Ethereum.

Pre-print copy of the manuscript published by publisher="Springer Nature Switzerland",
(available at `link.springer.com`)

Requirement	Our Solution	Metatransaction Batching/Relaying	General-State Channel Networks
R1: Minimize application latency	✓ Instant feedback with local blockchain simulation	✗ Increased per-transaction latency for a batch to accumulate	✓ Low-latency transactions through off-chain processing
R2: Minimize transaction fees	✓ Batching and selecting periods with lower gas fees	✓ Reduced fees through batching transactions	✓ Significantly reduced fees with only settling the final state
R3: Maintain full transaction history	✓ Full transfer of transactions to the public blockchain	✓ Full transfer of transactions to the public blockchain	✗ Only the final state is settled on-chain
R4: Guarantee transaction order and inclusion	✓ Nonce values and dispute resolution mechanism	✗ Can maintain order within a batch, but may not guarantee transaction inclusion by a batcher	✓ On-chain dispute process by publishing signatures of the state
R5: Provide a balanced approach to user self-custody	✓ Multisignature wallets provide convenience and security	? Varying degrees of user custody and traditional methods depending on implementation	? Often complex and may require higher technical expertise, less user-friendly

Table 1: Comparison with different solutions

Channel networks enable the participants to exchange simulated transactions off-chain, settling the final agreed-upon state on-chain. Examples are Bitcoin's Lighting Network [19] and Ethereum's Raiden [20]. General-state channel networks extend this idea to arbitrarily complex smart contracts [8]. In this approach, most of the transaction history is not recorded on the main chain; additionally, the parties need to be online to authorize new state transitions. While channels imply that multiple parties exchange simulated transactions among themselves based on self-enforcing signature authorizations, a local private blockchain can also be instantiated on the server of one party. Such a use-case has been explored in [21] for auctions.

To conclude, Tab. 1 qualitatively compares the proposed solution with the batching and general-state channel networks approaches while addressing the requirements elicited in Section 2.

6 Conclusion

This study presented a novel middleware solution that streamlines integration of public blockchains in hybrid architectures. By combining a local blockchain with asynchronous batch-transfers to a public blockchain, we ensure instant feedback and reduced costs. The effectiveness of the solution was confirmed by reduced latency and gas cost achieved by the Thankspay service prototype compared to two baseline cases. The solution also includes a dispute resolution mechanism enforcing accountability for the operator of the service.

At present, to ensure consistency between local and public blockchains, the owner can process only one transaction at a time. For instance, if we locally process a transaction with a higher positionNonce *before* a lower one is completed, it will be executed *after* the lower one on the public blockchain, leading to inconsistencies. In the future work, we plan to address this limitation, as well as to further enhance the dispute resolution mechanism and to test the system in more complex scenarios.

Pre-print copy of the manuscript published by publisher="Springer Nature Switzerland",
(available at `link.springer.com`)

References

1. Tai, S., Eberhardt, J., Klems, M.: Not ACID, not base, but salt. In: CLOSER 2017. (2017) 755–764
2. Wöhrer, M., Zdun, U.: Architectural design decisions for blockchain-based applications. In: ICBC 2021, IEEE (2021) 1–5
3. Ltd, P.P.: Blockchain: The solution for supply chain transparency (Nov 2015)
4. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: Medrec: Using blockchain for medical data access and permission management. In: OBD 2016, IEEE (2016) 25–30
5. Spain, M., Foley, S., Gramoli, V.: The impact of Ethereum throughput and fees on transaction latency during ICOS. In: Tokenomics 2019, Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
6. Donmez, A., Karaivanov, A.: Transaction fee economics in the Ethereum blockchain. Economic Inquiry **60**(1) (2022) 265–292
7. Wang, Y., Zhang, Q., Li, K., Tang, Y., Chen, J., Luo, X., Chen, T.: iBatch: saving Ethereum fees via secure and cost-effective batching of smart-contract invocations. In: ESEC/FSE 2021. (2021) 566–577
8. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: SIGSAC 2018. (2018) 949–966
9. McCorry, P., Buckland, C., Bakshi, S., Wüst, K., Miller, A.: You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies. In: FC 2019 Workshops, Springer (2020) 35–49
10. Sandford, R., Siri, L., Tirosh, D., Weiss, Y., Forshtat, A., Croubois, H., Tomar, S., McCorry, P., Venturo, N., Vogelsteller, F., et al.: ERC-2771: Secure protocol for native meta transactions (Jul 2020)
11. Chong, N.: As Ethereum Defi craze continues, Uniswap is processing over 100,000 transactions a day (Nov 2020)
12. Zhang, L., Lee, B., Ye, Y., Qiao, Y.: Evaluation of Ethereum end-to-end transaction latency. In: NTMS 2021, IEEE (2021) 1–5
13. Abadi, J., Brunnermeier, M.: Blockchain economics. Technical report, National Bureau of Economic Research (2018)
14. Jia, Y., Xu, C., Wu, Z., Feng, Z., Chen, Y., Yang, S.: Measuring decentralization in emerging public blockchains. In: IWCMC 2022, IEEE (2022) 137–141
15. Sguanci, C., Spatafora, R., Vergani, A.M.: Layer 2 blockchain scaling: A survey. arXiv preprint arXiv:2107.10881 (2021)
16. Seres, I.A.: On blockchain metatransactions. In: ICBC 2020, IEEE (2020) 178–187
17. Hughes, W., Magnusson, T., Russo, A., Schneider, G.: Cheap and secure metatransactions on the blockchain using hash-based authorisation and preferred batchers. Blockchain: Research and Applications (2022) 100125
18. Buterin, V., Weiss, Y., Gazso, K., Patel, N., Tirosh, D., Nacson, S., Hess, T.: ERC-4337: Account abstraction using alt mempool [draft] (Sep 2021)
19. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)
20. : Raiden network. Available at: `https://raiden.network/` Accessed: 15/06/2023.
21. Desai, H., Kantarcioglu, M., Kagal, L.: A hybrid blockchain architecture for privacy-enabled and accountable auctions. In: ICBC 2019, IEEE (2019) 34–43

References

Yessenbayev, Olzhas, Marco Comuzzi, Giovanni Meroni, and Dung Chi Duy Nguyen (2023). "A Middleware for Hybrid Blockchain Applications: Towards Fast, Affordable, and Accountable Integration". In: *Service-Oriented Computing*. Ed. by Flavia Monti, Stefanie Rinderle-Ma, Antonio Ruiz Cortés, Zibin Zheng, and Massimo Mecella. Cham: Springer Nature Switzerland, pp. 307–322. ISBN: 978-3-031-48421-6.

BibTeX

```
@InProceedings{10.1007/978-3-031-48421-6_21,
author="Yessenbayev, Olzhas
and Comuzzi, Marco
and Meroni, Giovanni
and Nguyen, Dung Chi Duy",
editor="Monti, Flavia
and Rinderle-Ma, Stefanie
and Ruiz Cort{\'e}s, Antonio
and Zheng, Zibin
and Mecella, Massimo",
title="A Middleware for Hybrid Blockchain Applications: Towards Fast, Affordable, and Accountable
Integration",
booktitle="Service-Oriented Computing",
year="2023",
publisher="Springer Nature Switzerland",
address="Cham",
pages="307--322",
abstract="Hybrid blockchain architectures combine centralized applications, like enterprise
systems, with (public) blockchain to implement additional functionality, such as tamper-proof
record keeping. To reduce the latency and cost of using a public blockchain, these systems
may rely on batching of transactions or general-state channel networks. While reducing
costs, the former increase the latency. With the latter, only major state updates are
recorded on-chain, while most transactions history remains only on the channels. This
paper describes a novel solution that combines the benefits of both approaches to decrease
the latency and cost of hybrid blockchain applications. We propose to combine a local
blockchain that runs on a centralized server to provide near-immediate state update confirmation,
with a batching mechanism sending transactions to a public blockchain for record-keeping
at a most convenient time. We also introduce a dispute mechanism promoting the prompt
delivery of correct batches to the public blockchain by the application provider, thereby
deterring malicious behaviours. The solution is motivated by a fintech use case, for which
we also show the implementation of a prototype and an experimental evaluation of the latency
and cost savings.",
isbn="978-3-031-48421-6"
}
```