Long-Term Quotaless Deterministic Batching on the UTXO Ledger

Quang Luong The Minh
quang@meld.com

Hai Nguyen Quang
hai@meld.com

Dat Duong Thanh
dat@meld.com

September 2021, Draft v0.2

Abstract

This paper presents a few eUTXO[1] smart contract architectures that allow long-term quotaless deterministic batching into a single state. Compared to a frequently updated reserve-based architecture, this class of solutions accepts more latency in exchange for zero congestion. The proposed architectures should cover most long-term on-chain states on any UTXO blockchain that supports scripting. We hope this work will lead to successful Cardano implementations and new smart contract architecture research in the future.

1 Introduction

Most smart contract designs so far have been done on an account-based ledger model like on Ethereum. Due to fundamental differences on the ledger level, many of these approaches cannot be ported to the UTXO world. With the rise of UTXO smart contracts on chains like Cardano, Ergo, and Nervos, there is fertile ground for innovative new smart contract architecture research waves.

This paper presents smart contract architectures that allow long-term quotaless deterministic batching into a single state. This work is essential in the UTXO world, where states are maintained in UTXOs that can only be spent once. Without a well-thought concurrent framework, there will be a race condition where different dApp users compete to consume the same UTXO. For example, multiple vote transactions consume the same proposal UTXO, but only one succeeds per block. Our architectures allow dApp users to create quotaless batch step UTXOs that are applied to the state UTXO with time. Compared to a frequently updated reserve-based architecture or a lazy update approach, this class of solutions accepts more latency in exchange for zero congestion and an unlimited number of batch steps.

2 The Architectures

2.1 Unordered One-Time States

One-time states finalize. One instance is a proposal that can be voted on, to be finalized as passed or rejected. The counterpart is continuous states with no deadline, like a liquidity pool that progresses with each swap.

An unordered state has commutative steps. For example, equal votes can be counted in any order without affecting the outcome of a proposal. A pre-order campaign can be a one-time state with order, where earlier pre-orders have better discounts.

2.1.1 Make Step

A step UTXO can be submitted directly by the user. The make-step transaction must mint an authentic token of the state and is guarded by its minting policy.

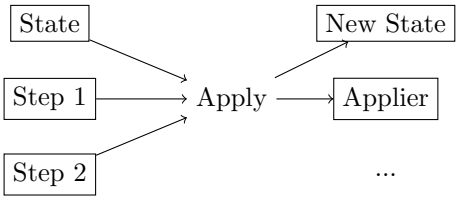
One-time states must have a make-step deadline to mark the latter state application phase. Think of the voting phase before the vote counting phase, the pre-ordering phase before the pre-order counting phase.

The authentic token's minting policy should guarantee that make-step transactions only validate before the deadline. The validation rule can be further extended depending on the application. For instance, a governance process might require burning an identity NFT to acquire a voting right. Another might require depositing stake tokens to determine the weight of the vote. Deposited tokens will then be returned when the step is applied.

At any given time, we have a set of unordered step UTXOs. Users can submit more steps in a quotaless and zero-congestion way.

2.1.2 Apply

Since order does not matter, we can apply steps in any way, any time. Off-chain agents can race to apply steps in a block for rewards.



...

More outputs might be produced depending on the application. For example, an applied vote can send its stake tokens to a time-locked script for the voter to withdraw after the voting deadline.

These non-deterministic & non-consensus architectures usually face a DoS attack where malicious agents submit quick transactions that apply only one step per block. To avoid this problem, the state should maintain a minimum factor m so that only transactions applying at least m steps validate. When there are fewer than m steps in a block, we skip it and wait for more in the upcoming blocks. One should find the largest m that fits the transaction size limit.

As order does not matter, it is straightforward for off-chain agents to apply faster by chaining transactions for more rewards. That said, there is a race condition among off-chain agents, so the minimum is still m steps per block.

2.1.3 Shrink

The state has to finalize after the make-step deadline eventually. A constant m does not work as we are stuck when fewer than m steps are left. This state is when we start shrinking m down.

The state should track the timestamp of each *Apply* transaction. When the last step application is one block away, the state starts accepting transactions with $round(\frac{m}{2})$ steps. m keeps shrinking to one, where the state accepts at least one step per transaction. The state can then finalize when no agents could apply anything in the previous block.

dApp developers can set how fast m shrinks. For example, m can shrink straight to one for a proposal that executes on-chain in the further future. At worst, the state finalizes in m blocks since shrinking, which is half an hour with a previous m of 90 and a block duration of 20 seconds. This latency is acceptable when the proposal needs preparation time before execution. For instance, a proposal that whitelists a new DeFi asset might wait a week for its oracle feed to get stable on-chain first. When such a DoS attack is meaningless, and the net profit from a one-step transaction is close to zero, off-chain agents are incentivized to apply more and faster.

2.1.4 Final notes

This architecture guarantees that no steps are left behind without reserves, predefined limits, shared states, or user congestion. To avoid unwanted timing issues, a state can wait for more than one non-apply block before finalizing.

Even when there is not much a malicious agent can do, dApp developers should run honest bots themselves to ensure constant progress.

All steps are transparent and verified by the minting policy of the authentic token. Anyone can track the whole progress and prepare accordingly. For example, a non-profit foundation can adapt plans and start acting as they confirm on-chain donations for their charity.

The final state is known when the make-step deadline ends; finalization is only required for further on-chain processing. The outcome can be minting an NFT medal, updating on-chain protocol parameters, transferring on-chain funds, and more.

2.2 When Order Matters, Dispute!

When order matters, we need a mechanism to ensure that all steps are applied in order. We do this by making off-chain agents race to announce a batch for the rest to double-check and dispute. The dispute phase blocks and a batch of steps can only be applied to a state when the previous block has no dispute. One can easily dispute a malicious batch by showing a step that is ordered earlier than the last step of the announced batch.

Off-chain agents have to deposit collateral to announce a batch and lose it when successfully disputed. This requirement makes DoS attacks on long-term states very expensive. Moreover, malicious agents can only sacrifice their capital for state application latency; users always use quotaless to create steps.

2.2.1 Make Step

A step UTXO can be submitted directly to every block with a timestamp t for ordering. The make-step transaction must mint an authentic token of the state and is guarded by its minting policy. The supplied timestamp is padded to a range r and provided to an on-chain `MustValidateIn` constraint. This rule requires the transaction's validity range to be contained in r. A dishonest t will fail to make the step.

For example, it is now the 26th second globally, and the user honestly says 26 in the redeemer. This timestamp is then padded with 3 seconds to construct the range r (26, 29). If this transaction is validated by the 29th second, the step passes. If the user lies with 20 to front-run other steps, it will construct the range r (20, 23). This duration has already occurred before submit time, so a transaction with a validity range in it would not validate.

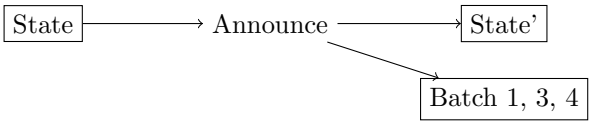
Note that it might still be possible to front-run a step by copying the step but with a $t-1$ timestamp. We are still studying this race condition to see if we can prevent it or provide the best formula for everyone to follow. The intuition is that if we pad too much, a later step registering an earlier t will break the order; if we pad too tight, honest steps might fail due to race conditions.

At any given time, we have an ordered set of step UTXOs. Users can submit more steps in a quotaless way.

2.2.2 Announce

In the *Announce* phase, off-chain agents race to announce a batch UTXO that declares all unannounced steps. The state UTXO should be included to transition to the next phase, which also guarantees only one active batch to simplify the following processes. This way, a validator script can also support many states, as the authentic token name t can be separated and defined per state.

An *Announce* transaction announces a batch of ordered unannounced steps and transitions the state to the *Dispute* phase.

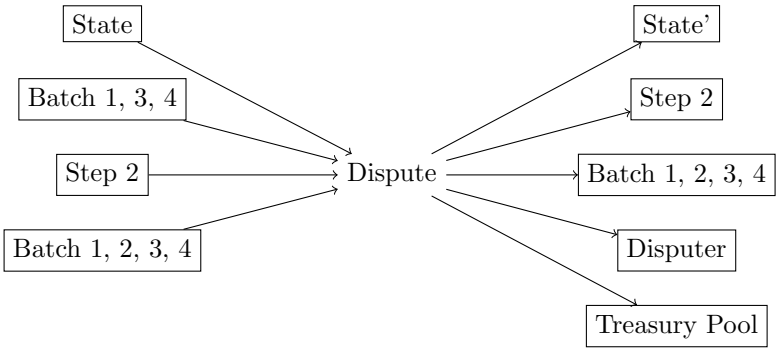


A batch UTXO should hold collateral and a public key to receive the collateral back with rewards when it is proven honest. If the batch is dishonest and disputed, the collateral will be split among a treasury pool and the disputer. One should send more to the treasury pool to avoid a DoS attack where malicious agents dispute themselves to cycle capital. Note that the state UTXO can act as a treasury pool for itself.

A batch can include an unlimited number of steps in its datum because only the datum hash is stored on-chain. This property allows us to enforce a single correct announcement that is all the remaining steps up to the previous block. A single correct batch then prevents the one-step-per-batch DoS attack where malicious agents submit the single earliest step, which is orderly correct.

2.2.3 Dispute

After a batch announcement, the state transitions to the *Dispute* phase. A transaction can *Dispute* a batch UTXO by providing a step that was made earlier than the last step of the batch. If a *Dispute* succeeds, the batch collateral is collected for lying, and the state transitions back to the *Announce* phase. Alternatively, the disputer can submit a batch to keep the *Dispute* phase and reduce unnecessary latency from reverting.



A malicious batch might declare a non-existing step UTXO, or an existing step with a wrong datum. Another *Dispute* redeemer with collateral is required in this case, to dispute that a declared UTXO is malicious. Anyone can then counter-dispute by consuming the UTXO in question. If such an honest UTXO exists, the original disputer loses the collateral. If there is no counter-dispute

after a number of blocks, the original batch is successfully disputed for declaring a malicious UTXO.

Until read-only UTXOs are supported, included step UTXOs change after a dispute. This inconvenience requires identification by the datum hash, which should hold an identification like the public key of the step maker.

Note that the collateral penalty is put on the off-chain agents that submit the batch UTXOs instead of the users. It is then costly for attackers to announce a malicious batch, especially when other agents can swiftly find a left-out or wrong step for a dispute. As these batching agents are more advanced and involved, we can push the collateral relatively high without affecting the regular users.

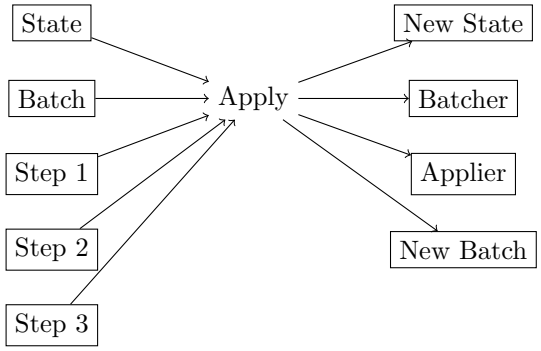
2.2.4 Apply

A state can *Apply* its batch if the *Batch* time was over a block away, meaning no one could dispute the batch in the last block.

As a batch can include an unlimited number of steps in its datum, a state should consume up to only s states per transaction depending on the transaction size limit. It should also maintain an index to keep applying through the batch. Once a batch is applied, the state can continuously apply until its end without further dispute.

An *Apply* transaction should return the collateral to the batcher, as well as rewarding the agents that *Batch* and *Apply*. For sustainability, these rewards should be accumulated from each user step. Extra rewards are possible when the state acts as its own treasury pool.

When a batch is fully applied, the state can transition back to the *Announce* phase. Alternatively, to reduce latency, the final applier can announce the next batch right in this phase.



...

The state then only needs to get back to the *Dispute* phase. It is usually better to enforce announcement with state application, so DoS attackers do not race with a fast *Apply* transaction. A dishonest applier that announces an empty or minimal batch can be disputed in the next block.

2.2.5 Final Notes

It is to be noted that timing often comes from a non-absolute off-chain timestamp. It might not be trivial to tune time duration in terms of a block. A state can wait for more than one non-dispute block before applying a batch to avoid unwanted timing issues.

Furthermore, there is still a distributed race condition to solve to prevent front-running.

Malicious agents are only nullified and punished when there is an honest agent that can submit every block. dApp developers should run distributed honest agents themselves.

Apart from that, this quotaless architecture enables deterministic batching into a continuous state that does not require fast state application. Users do not have to compete for reserves or worry about malicious ordering and omission.

This architecture works for both one-time and continuous states. It also works for unordered steps, as we can always add a timestamp to each.

3 Experiments

We are prototyping multiple staking and governance contracts using these architectures:

- Concurrently stake MELD tokens to the insurance pool.
- Concurrently vote to whitelist new mintable fiats for the MELDed Fiat service.
- Concurrently cast a billion votes on a proposal.
- Low-latency concurrency on a dedicated sidechain.

We have collected valuable findings along the way, especially on more techniques that work better in a few specific instances. Few timing issues have arisen in Layer-1 too. We hope to demonstrate them in the upcoming versions and papers.

After a few different working prototypes, we will start generalizing the architectures into a library that can be reused anywhere. We might need to use Template Haskell to keep as many components reusable as possible. Library users can then focus on writing only their application-specific logic. This endeavor should also yield more exciting ideas for us to raise and implement on Cardano public repositories.

We intend to open-source the library as soon as we reach a stable state.

4 Related Work

4.1 Batching

Our architecture belongs to this group of solutions. The main advantages we have identified at this time are:

- *Security.* Malicious agents are nullified by default or when there is an honest agent that submits in a block. Developers should run distributed off-chain agents for their own dApps. It is trivial to maintain these bots, especially when compared to maintaining a front-end with inbound traffic.
- *Simplicity.* The architecture is well-scoped. The off-chain infrastructure is minimal.
- *Quotaless.* There is zero congestion for end users. There is no limit on the number of steps, only on the capital required to make them.

There are still many exciting ideas that we can learn from other solutions, especially to reduce latency without sacrificing the quotaless and zero-congestion trait.

4.2 Divide & Conquer

We can break a state UTXO into smaller UTXOs for concurrent interactions. For instance, multiple voting boxes can be concurrently voted on every block. These boxes can then be combined into a final state after the voting duration.

This solution may work for low-traffic states. Otherwise, it has a limit on the number of concurrent steps per block. Congestion can also happen to users interacting through different interfaces that happen to choose the same state UTXO. It may not work when sub-states need a shared state as well.

It is to be noted that this solution is not exclusive to our architectures. For example, we can have s sub-states for a high-traffic unordered one-time state. When there are way more than $s * m$ steps, shuffling agents might be able to apply close to $s * m$ steps per block.

4.3 Lazy Update

Lazy update is another line of techniques that we have been studying. The idea is to reserve to create update UTXOs. However, instead of constantly applying the update steps into a state, we wait until such a state is required on-chain.

For example, in a governance voting process, we only need to consume the votes when we consume their proposal to decide if it passes or not. Constantly applying votes to the proposal requires more time and fees. The off-chain world is unaffected, as it can still find the vote UTXOs and show users the state of the proposal.

That said, this approach is usually not quotaless and has congestion for end users. Early batching is required, regardless, when there are too many steps to be applied at once at the end.

4.4 Auto-Scaling Concurrent & Deterministic Batching on the UTXO Ledger[3]

This is another line of concurrency research that we have been pursuing. It is reserve-based and allows constant deterministic step applications. This architecture suits frequently updated states like swapping on a liquidity pool. The main trade-off is congestion when users fight for the same reserve. Latency may also arise when there are not enough reserves to handle high-traffic usage.

4.5 Layer-two Solutions

Layer-two solutions like Hydra[2] are significant for this concurrency and scaling challenge. While the presented architecture could work on layer-one, we suspect it would work even better on layer-two. We are looking forward to experimenting with this architecture on a dedicated MELD sidechain for better timing with just MELD transactions.

5 Conclusions

This paper presented a few solutions to the concurrency problem on UTXO ledgers, especially for on-chain states that do not require instant finalization. These architectures are still in their early forms. More specialized techniques are needed to scale different dApps further. We have, for example, been designing extended and different architectures for different MELD components.

We have a massive backlog of innovative ideas to explore and expect to publish many new versions and research efforts soon. It is now indeed the best time to do R&D on Cardano and the UTXO ledgers in general.

References

[1] M. Chakravarty, James Chapman, K. Mackenzie, Orestis Melkonian, M. P. Jones, and P. Wadler. The extended utxo model. In *Financial Cryptography Workshops*, 2020.

[2] Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, A. Kiayias, and A. Russell. Hydra: Fast isomorphic state channels. *IACR Cryptol. ePrint Arch.*, 2020:299, 2020.

[3] Dat Duong Thanh and Hai Nguyen Quang. Auto-scaling concurrent & deterministic batching on the utxo ledger. 2021.