Auto-Scaling Concurrent & Deterministic Batching on the UTXO Ledger

Dat Duong Thanh
dat@meld.com

Hai Nguyen Quang
hai@meld.com

September 2021, Draft v0.5

Abstract

This paper presents an eUTXO[1] smart contract architecture that allows auto-scaling concurrent and deterministic batching into a single state. Our solution is entirely decentralized and does not depend on honest or incentivized off-chain agents. We also cover more advanced topics like DoS prevention, front-running, and MEV. While this paper uses batching swap orders into a liquidity pool on Cardano as the case study, the architecture should adapt to any frequently updated on-chain state on any UTXO blockchain that supports scripting. We hope this work will lead to a successful Cardano implementation 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 a smart contract architecture that allows auto-scaling concurrent and 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 swap transactions consume the same liquidity pool UTXO, but only one succeeds per block. Our architecture allows dApp users to create multiple batch step UTXOs that are applied to the state UTXO in one single transaction. The key is to have a deterministic validation rule on-chain that enforces deterministic inputs and outputs. With that, the agent submitting this transaction cannot omit any input, rearrange the order, or change the output.

2 The Architecture

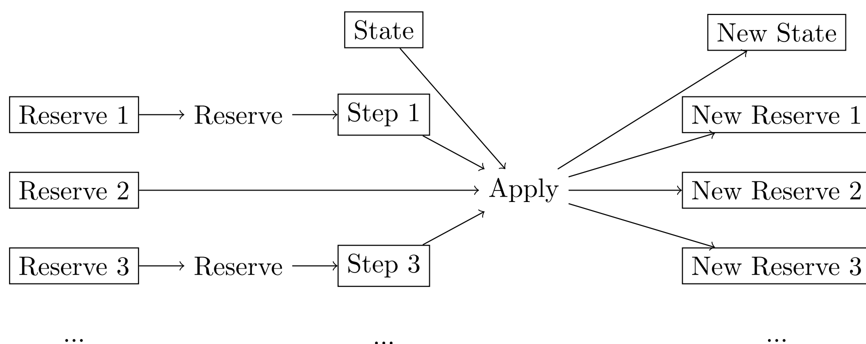
2.1 Overview

We start with a state UTXO that we want to apply batch steps to. We then attach a concurrency number c to its datum to determine the maximum number of batch steps it can apply in a single transaction. Every time a state UTXO is created, c reserve UTXOs must be created along with it.

A user can *Reserve* a step UTXO by consuming a reserve UTXO. A step UTXO must contain all data and values required to apply the step to the state. For a simplified liquidity pool interface, a step holds the public key of the trader and the tokens of one pair to be swapped for the counterpart in that pair.

After the *Reserve* phase, anyone can *Apply* the steps into the state. The validation rule of *Apply* requires c corresponding reserve and step UTXOs as inputs. This disallows the transaction from omitting any reserved steps from any user. The validation rule must be deterministic to guarantee a single set of outputs for a given set of inputs. For instance, this would disallow the transaction from choosing a preferred end-state by flagging a reasonable step as invalid or mixing up the ordering to front-run on a liquidity pool.

The outputs contain the new state and c new reserve UTXOs. Depending on the application, there may be user outputs such as a successful swap with the desired tokens or a failed swap with the original tokens. Alternatively, a failed step can be configured to remain for the next *Apply* transaction. After a certain amount of time, a step can be rejected back to a reserve.



We continue to explore key requirements and details of the architecture in the following subsections.

2.2 Core UTXOs

2.2.1 State

The core component in the architecture is the state that we need to consume UTXOs against concurrently. An example of this is a liquidity pool that transitions per swap order.

While the content of the state is application-dependent, there are a few key pieces of information to have:

- A concurrency level c: This is the total number of reserves and steps that exist at any given time, hence the maximum number of batch steps that can be applied in a transaction. This should be set to a number that meets the current transaction size limit.

- A queue of past steps per block q: This queue can be used to auto-scale the concurrency level c for the next block. For every *Apply* transaction, we can use the current number of steps s with all the past number of steps from q to set c for the next block, dequeue the front item of q, then enqueue s to q. The scaling function is application-dependent, and it is usually better to scale down slower than it is to scale up. For example, when q is `[10, 9, 8, 7]` and the current s is 6, we can create 9 or 10 reserves for the next block, even when the original c is 50. c will be raised again when the next s raises.

- A number of past steps to track p: This is simply the size of q to maintain. An application can have a higher p to have more past data for a smoother scaling process.

- A range for the concurrency level r: This defines the minimum and the maximum number of c to clamp. This range is useful to make sure c will not scale too high to cross the transaction size limit or scale too low to have congestion when a large group of users gets back at the same time.

- A token name t: This is a unique token name that determines the reserve and step UTXOs that belong to this state. For example, a DEX might have a currency symbol for all liquidity pools, with a unique token name for each specific pair. These tokens can only be minted with the state UTXO.

- A validity interval i: This is used to organize transaction validity intervals, which helps organize the *Reserve* and *Apply* phases. To prevent users from consuming UTXOs with the wrong redeemer, which would lead to congestion.

- A binary era e: This can be used to organize phases further. For instance, in era 0, the state is applied with steps from era 0, creating new era 0 reserves before turning into an era 1 state, with era 1 reserve UTXOs getting reserved in the same block. This allows state application in every block and effectively requires c era 0 reserves plus steps and c era 1 reserves plus steps at any given time.

dApp developers can freely configure the architecture as fit for each of their states. For example, a simple state that does not need extra datum storage for auto-scaling can go with a constant c to be updated through a decentralized voting process when needed.

The validation rule is also application-dependent, with a few foundational rules:

- Apart from the state itself, both the input and output sets must contain exactly c UTXOs, each with exactly one t token. This rule ensures deterministic input sets so that no user steps are left behind.
- The validation rule has to be deterministic. The same set of inputs must always result in the same set of outputs. This rule prevents any influence from off-chain agents that submit the *Apply* transaction. When step order matters, we can sort by a step attribute to still apply deterministically. For a liquidity pool, this can be the approximate submit time of each swap order step.

A state UTXO may also store values, like tokens of both pairs, for a liquidity pool.

2.2.2 Reserve

A reserve is a UTXO that dApp users consume to produce a Step UTXO. It is used to guarantee that no steps are left behind in an *Apply* transaction.

If users can create steps directly without going through this reserve process, steps will be left behind when there are more steps than a transaction size limit can include. Furthermore, malicious off-chain agents can omit steps by choice because the on-chain state is not notified when each step is created.

A reserve is application-dependent, with a few foundational rules:

- Store a t token for validation against its corresponding state and step.
- Have a binary era e that can be used to organize non-blocking phases.
- Have enough data to create a valid step. For example, a liquidity pool reserve should know what assets can be deposited to create a swap order step and a potential limit on the amount.
- Have an approximate creation time to prevent an *Apply* consumption during the *Reserve* phase and vice-versa.

2.2.3 Step

A step is a UTXO that dApp users create to apply to the state UTXO. For instance, this can be a swap order for a liquidity pool.

A step is application-dependent, with a few foundational rules:

- Store a t token for validation against its corresponding state.
- Have a binary era e that can be used to organize non-blocking phases.

- Have enough data to make a valid state transition. For example, a liquidity pool step should store the deposited assets and attach the swapped assets to a Public Key.
- Have enough data to ensure a deterministic state transition. For instance, a liquidity pool step should have an approximate reserve time for the steps to be applied in that order.

2.3 Timing

Timing is vital to address the congestion problem of the UTXO model. For example, we cannot allow a *Reserve* consumption of a reserve UTXO when an *Apply* transaction also happens for that reserve in the same block.

To solve this problem, we add a validation interval i to the state to control the validation interval of created reserve UTXOs. Before a certain threshold, only a *Reserve* transaction can consume a reserve. After that threshold, only an *Apply* transaction can consume a reserve. This technique creates two non-conflicting phases to avoid congestion.

To avoid the one-block-one-phase problem, we can introduce a binary era e to the state. For instance, in era 0, the state is applied with steps from era 0, creating new era 0 reserves before turning into an era 1 state, with era 1 reserve UTXOs getting reserved in the same block.

Note that timing often comes from a non-absolute off-chain timestamp. It might not be trivial to tune time duration in terms of blocks.

Validation intervals and timing are not just great tools to fight congestion; they can also be utilized for deterministic validation.

2.4 Deterministic Validation

Deterministic validation is the key characteristic that prevents any influence from off-chain agents. Any state should have a deterministic validation rule that can be shared both on-chain and off-chain.

That means designing a pure transition function to apply to the state with each step iteratively.

$$f :: (State, [Output]) \to Step \to (State, [Output])$$

Each iteration should validate the step. If it is valid, update the state and append the corresponding valid output to the `Output` list. If it is not, append the corresponding invalid output to the `Output` list.

When ordering affects determinism, we have to sort the steps through an attribute before the iterative fold. For example, ordering matters for a liquidity pool state because it changes the price after each swap. Also, the off-chain agent can select any of the two similar steps that mark the end of liquidity for further swaps without order. Certain applications like voting might accept these problems when order does not matter. Others will have to find a unique attribute for each step to sort on.

A powerful attribute that can be assigned to most cases is the approximate reserved time t of each step. We can acquire it through an off-chain timestamp supplied in the *Reserve* transaction. t is then 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 reserve.

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 validates by the 29th second, the reserve passes. If the user lies with 20 to front-run other steps, it will construct the range r (20, 23). This duration has already gone at 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 reserve 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, there will be much room for front-running; if we pad too tight, honest reserves might fail due to race conditions. Commutative steps do not have such a problem.

With this deterministic validation rule, the on-chain script first calculates the outputs, including the new state, then compares them to those supplied through the transaction. Only transactions that provide this correct answer validate. Then off-chain agents can only use the same validation rule to construct the only valid *Apply* transaction.

Since this deterministic validation rule is shared both on-chain and off-chain, it should be possible to tell if a step would validate at submit time. This rule helps prevent users from paying for steps that would fail. If the race condition at submit time is acceptable, dApp developers can also utilize this property to punish steps that fail to progress the on-chain state, as a DoS suspect that fails deliberately to avoid a fee like a yield for a successful swap.

2.5 Scalability

The architecture's scalability is very well-defined through the concurrency level c in the state. Since up to exactly c steps can be applied to a state in a single transaction, we can update this number to fine-tune the concurrency level. This can be done automatically through an auto-scaling mechanism or voted manually through a decentralized voting process.

It is essential to set c and the upper bound of r low enough not to risk crossing the transaction size limit. dApp developers need to monitor ledger transaction size updates to adapt in time.

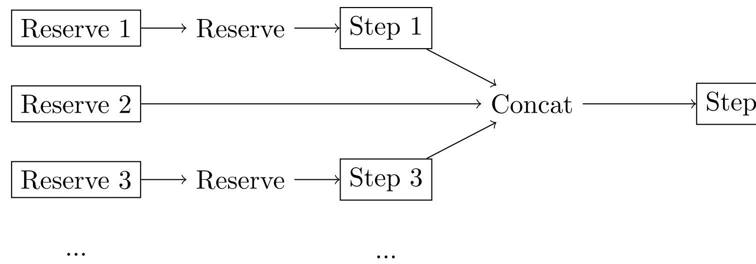
As each state has its own concurrency level, we can have a high c for liquidity pools with higher traffic and a low c for those that do not. Low-traffic states can keep c low to reduce transaction fees and the number of unused UTXOs.

It is to be noted that our architecture is a general solution to a big problem. More specialized techniques might be needed to scale different dApps further. We also mention more dimensions to scale in the Related Work section.

2.6 Concating Steps

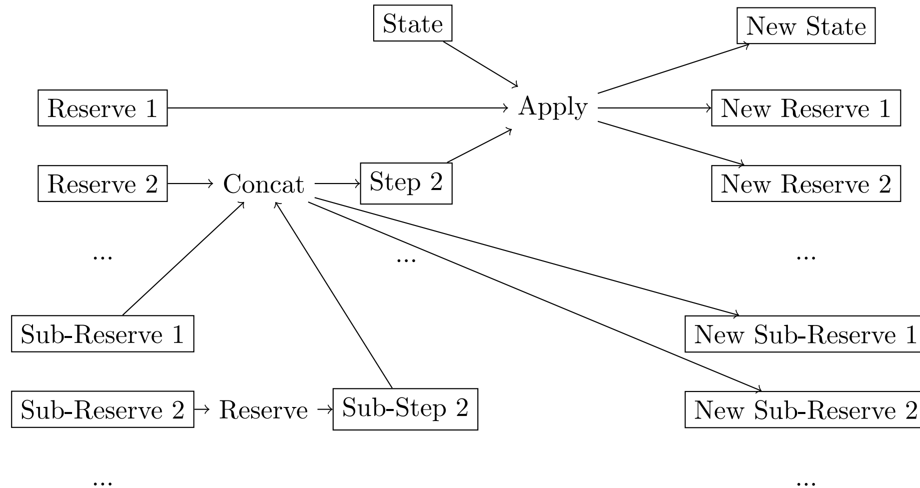
It is a lot more space-efficient to work with a UTXO of steps than many step UTXOs. The reason comes from the overheads that come with each UTXO. For example, all step UTXOs share the same validator hash. Combining multiple step UTXOs into a UTXO of steps reduces a lot of duplicated validator hashes in the *Apply* transaction. A larger Datum also does not yield a larger hash.

Therefore, if we can make a step a Monoid, with a reserve as the `mempty`, and a `mconcat` that concats steps together, we can create another concating phase to trade a block worth of latency for more throughput.



Since the t tokens are simply accumulated after concating, the validation rule of the state simply needs to count for exactly $c\ t$ tokens. As this new phase is also non-blocking, we can promote the binary era e to a ternary era e to guarantee an *Apply* every block.

Alternatively, we can design a subsystem that concats steps into reserves.



This architecture also requires an additional phase from the user sub-step getting into a concatted step before an *Apply* transaction but requires more UTXOs and processing. For now, it is an attempt to complicate the architecture in search of new insights. We do plan to introduce more data structures in the upcoming draft versions.

2.7 UTXO Discovery

For a specific dApp with a specific endpoint like `https://api.dapp.com`, it is straightforward for its backend to track all reserves, steps, and state UTXOs of a single state. This can be done by tracking all UTXOs that store the t token, with a minting policy that guarantees these UTXOs to store it at any given time, and only those can.

To avoid congestion, this backend can queue requests so that no two requests target the same UTXO in the same block. Congestion might still happen from external users, which is hard to prevent in a fully decentralized system. Luckily, this only happens for reserving the same UTXO during the *Reserve* phase. Steps and states only have one deterministic *Apply* redeemer in the *Apply* phase. We will continue to research a solution for this specific congestion. A solution that exchanges latency for no such congestion is presented in Long-Term Quotaless Deterministic Batching[4].

At the time of writing, we recommend customizing the Chain Index component of the Plutus Platform for UTXO discovery and management.

3 Experiments

We are prototyping multiple smart contracts using this architecture:

- Concurrently mint MELDed Fiat like mEUR against a governance state that tracks what fiat can be minted and collateral prices. Concurrently deposit/withdraw collateral, mint/burn mFiat at a CDP against a single oracle state that tracks prices.
- Concurrently lend and borrow against a MELD lending pool.
- Concurrently stake MELD to a governance state for validating governance proposals.
- Concurrently apply swap orders against a single MELD Vault (single-sided liquidity pool).

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 architecture 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 this 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:

- Deterministic on-chain validation rule. Off-chain agents can only submit the only correct output. This rule means no influence whatsoever from any off-chain agents.
- Security. No edge exploitation from off-chain agents.
- DoS resilient. Since the off-chain batching agent has no influence, a DoS attacker would need to consume as many reserves per block as possible. If the small steps are legit, they are still net gains for the dApp. This DoS attack is also not sustainable as attackers have to pay transaction fees for each reserve and yield for a swap. We can also punish steps that deliberately fail and do not progress the state with deterministic validation off-chain. Users have to pay for the steps for other solutions that give power to the batching agent, while the malicious agents still get rewarded for such small steps. Also, it does not matter if this architecture is under attack or not; it still guarantees up-to auto-scaling c steps per block. In contrast, a successful one-step transaction would render other solutions one step per block. dApp developers with this architecture can also set a minimum limit (like minimal swap value) to make it more expensive for DoS attackers to reserve. While on other solutions, it does not matter if such a limit or cost structure exists; as long as there is a legit order that a user has paid for, the one step per block DoS attack is there.

  We are researching more data and cost structures to make it even more expensive for attackers to pay for reserves. We will include dedicated sections for DoS prevention and cost structures in the upcoming paper versions.
- Capital efficiency. We do not have to pay more for off-chain agents to act with good intent.
- Simplicity. The architecture is well-scoped. The off-chain infrastructure is minimal. We do not have to design complex incentive functions that may be affected by market conditions and require further governance to tune right.

There are still many exciting ideas that we can learn from other solutions, like chaining transactions in the same block. Since the transaction size limits our architecture, more extended dimensions to scale are still desired.

4.2 Divide & Conquer

If only one state UTXO can be consumed per block, we can break it down into smaller UTXOs to be spent concurrently. For instance, we can have multiple lending pools for lenders and borrowers to interact with per block.

This solution, however, is usually not capital efficient. One lending pool might not have enough assets for a big borrower to borrow from. When lenders want to withdraw their supplied assets, they have to consume all lending pool UTXOs they have lent to. Both these cases require consuming multiple UTXOs to process a request, which requires more UTXOs to avoid congestion, which again lowers the number of assets at each pool. It is not trivial to solve this balance.

This design also only works for applications with no shared state among the sub-states. It is tough to fit applications that do and those that require synchronization or accumulation of the sub-states.

Not all states contain capital to suffer from this problem. In certain applications like oracle price consuming, it may be plausible to have multiple similar datum-only UTXOs.

Note that this design is not exclusive to our architecture. One can break down a state into smaller states, then use our concurrent architecture on each sub-state. Since the maximum size of a transaction limits our architecture's concurrency level, this approach can add another dimension to scale.

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 like the batch steps in this architecture. However, instead of constantly applying the updates 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 state risks congestion and unnecessary transaction fees. The off-chain world is unaffected, as it can still find the vote UTXOs and show users the state of the proposal.

Another instance is to store just enough data to calculate the interest a borrower has to pay at any given time instead of constantly updating this interest on-chain.

The key questions to ask when designing on-chain states are then:

- When do we have to consume the state on-chain?
- How often do these events occur?
- How do we design the contract to do less on-chain and do more off-chain?
- How do we design the deterministic validation rule and transition function for the state?

4.4 Long-Term Quotaless Deterministic Batching[4]

This is another line of research that we have been pursuing. It is not reserve-based and lets users create steps directly to guarantee zero congestion. The main trade-off is latency, as we need more blocking state phases to guarantee no steps are left behind.

For one-time states with no order, we keep submitting steps and batching them into a state until the make-step deadline. We then shrink the minimum number of steps needed to be applied per block until it reaches one to ensure all steps are applied. For example, we keep applying at least 64 steps per block until a block where no transaction consumes the state. Then starting from the next block, we apply at least 32 steps, repeat until the last step. A malicious agent cannot affect an honest agent in this scenario.

For continuous states with an order, we allow off-chain agents to dispute others and punish the malicious ones. 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 dispute a malicious batch by including a step ordered earlier than the last step of the batch.

We are looking for solutions that better balance quota and latency, a class of architectures that brings the best out of both these two.

4.5 Purely Functional[2] Designs

This paper was inspired by the difficulty of implementing state-centric models from account-based ledgers on a UTXO ledger. While studying workaround solutions to support those state-centric models, we suspect that more elegant UTXO-driven solutions exist for different applications.

For example, we have been designing a market model called One Shot Markets[5] where we bring price discovery off-chain to keep the on-chain markets pure and independent. It can be considered as a combination of an on-chain limit order book and an off-chain AMM. An on-chain swap is then a transaction that consumes a set of markets to create more markets. This design builds on strong parallelism of the UTXO model, where different outputs can be spent independently in the same block. There is no bottleneck if there is no shared state among the UTXOs.

4.6 Layer-two Solutions

Layer-two solutions like Hydra[3] 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 requiring constant updates. The architecture is still in its early form. 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 Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompson, Rebecca Valentine, and Philip Wadler. Functional blockchain contracts. 2019.

[3] 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.

[4] Quang Luong The Minh, Hai Nguyen Quang, and Dat Duong Thanh. Long-term quotaless deterministic batching on the utxo ledger. 2021.

[5] Hai Nguyen Quang and Tuan Tran Quoc. One-shot markets: Redefine liquidity on the utxo ledger. 2021.