Do we need chained transactions?

Aztec already supports consuming notes generated earlier within the same transaction - so called"transient reads".

Chained transactions

would extend this functionality to consuming notes generated by previous transactions that have been included in a block, but not yet confirmed/finalized

. Right now, the dependent notes cannot be proven in the private kernel circuit until the finalized note tree is updated, which could potentially take ~10 minutes.

We could add a pending_read_requests

array to the private kernel inputs, and modify the rollup kernel to have a rolling, up to date note tree. This would allow the existence of the pending notes to be verified inside the rollup kernel, unblocking the submission of the downstream transactions. This is primarily relevant for repeated interactions with the same contract, since notes are "siloed" across contracts.

As currently specified, we see the following tradeoffs in adding this feature:

Pros:

Eliminates wait time for certain transaction patterns

A user will now be able to submit multiple dependent transactions quickly, without noticing the underlying "time to finalize/prove" a block.

Example use cases:

Enables users to "checkpoint" a series of private transactions, possibly deferring public calls to a final chained transaction to defend against possible reversion in the public environment.

Enables users to "checkpoint" clientside proving. They can avoid the situation of a single function call proof fails inside a transaction proof, so a smaller amount of generated proofs are wasted. Under chained transactions, only the "downstream" transactions would be blocked, but transaction upstream of the proof failure would remain valid and submittable.

Allows the user to get around the maximum number of notes that can be created by a single transaction, when interacting with the same contract, without having to wait.

Cons:

More complexity in Sequencing/Mempool eviction

Transactions now form a DAG, so the sequencer and mempool need extra logic for checking for cycles/validity of declared dependencies and reconstructing the dependency graph before submission.

Additional complexity in rollup circuits

Rollup circuits already maintain rolling note tree roots needed to support pending transaction reads, but we would need to add membership checks to the circuits. We are already considering having support multiple circuit variations for handling e.g. different length array inputs. Simpler circuits would help with security and audibility.

Additional Cross wallet synchronization

There is also potentially now synchronization that must happen across a user's wallets on different devices, and between different users (if we support cross user transaction chaining.)

Not the only solution for the desired UX outcome

The frequency at which users would need to chain transactions can be reduced - we list some approaches below.

Alternative Solutions

We needed chained transactions in Aztec Connect because (1) there was a single global contract, so all transactions were inherently blocking and (2) proof/block time was several hours.

In Aztec, we're not sure whether this will situation be more of an edge case or more common situation for users, but believe we can try to address the UX benefits of chained transactions in other ways.

1. Smart contract level accommodation
2. Smart contract developers can try to reduce the need for chained transactions by more flexibly allowing consumption of notes within a transaction, or providing “orchestrator” methods for consuming notes within a same transaction, as is already supported.
3. If the base smart contract is inflexible, secondary “coordinator” or “aggregator” contracts could combine multiple calls into the same transactions.

4. Smart contract developers can try to reduce the need for chained transactions by more flexibly allowing consumption of notes within a transaction, or providing “orchestrator” methods for consuming notes within a same transaction, as is already supported.
5. If the base smart contract is inflexible, secondary “coordinator” or “aggregator” contracts could combine multiple calls into the same transactions.

6. Optimistic Proof generation by wallet
7. Once upstream transaction(s) have been sequenced, a wallet could optimistically generate the new note inclusion proofs against an unfinalized note root, and submit the transaction for the new L2 block. Pre-confirmations would help here.

8. Once upstream transaction(s) have been sequenced, a wallet could optimistically generate the new note inclusion proofs against an unfinalized note root, and submit the transaction for the new L2 block. Pre-confirmations would help here.

9. Faster L2 block times
10. This complements (2); the faster we can reliably assume upstream transactions are included and can be be proven against, the shorter the window for requiring chained transactions as opposed to “optimistic” transactions.

11. This complements (2); the faster we can reliably assume upstream transactions are included and can be be proven against, the shorter the window for requiring chained transactions as opposed to “optimistic” transactions.

12. Increase the maximum number of notes created by a transaction.
13. If a portion of transaction fees is linear in the number of notes created, we can still deter spam and allow what would have required multiple transactions to be executed in a single transaction. This is a partial solution, as it potentially allows chained transactions to be merged into a single transaction, but does not address the “checkpoint” type use cases mentioned above.

14. If a portion of transaction fees is linear in the number of notes created, we can still deter spam and allow what would have required multiple transactions to be executed in a single transaction. This is a partial solution, as it potentially allows chained transactions to be merged into a single transaction, but does not address the “checkpoint” type use cases mentioned above.