Transaction Confirmation & Expiration

Problems relating totransaction confirmation are common with many newer developers while building applications. This article aims to boost the overall understanding of the confirmation mechanism used on the Solana blockchain, including some recommended best practices.

Brief background on transactions#

Before diving into how Solana transaction confirmation and expiration works, let's briefly set the base understanding of a few things:

- what a transaction is
- the lifecycle of a transaction
- what a blockhash is
- and a brief understanding of Proof of History (PoH) and how it relates to
- blockhashes

What is a transaction?#

Transactions consist of two components: amessage and alist of signatures . The transaction message is where the magic happens and at a high level it consists of three components:

- alist of instructions
- to invoke,
- alist of accounts
- to load, and
- a“recent blockhash.”

In this article, we’re going to be focusing a lot on a transaction’srecent blockhash because it plays a big role in transaction confirmation.

Transaction lifecycle refresher#

Below is a high level view of the lifecycle of a transaction. This article will touch on everything except steps 1 and 4.

1. Create a list of instructions along with the list of accounts that
2. instructions need to read and write
3. Fetch a recent blockhash and use it to prepare a transaction message
4. Simulate the transaction to ensure it behaves as expected
5. Prompt user to sign the prepared transaction message with their private key
6. Send the transaction to an RPC node which attempts to forward it to the
7. current block producer
8. Hope that a block producer validates and commits the transaction into their
9. produced block
10. Confirm the transaction has either been included in a block or detect when it
11. has expired

What is a Blockhash?#

A“blockhash” refers to the last Proof of History (PoH) hash for a“slot” (description below). Since Solana uses PoH as a trusted clock, a transaction’s recent blockhash can be thought of as atimestamp .

Proof of History refresher#

Solana’s Proof of History mechanism uses a very long chain of recursive SHA-256 hashes to build a trusted clock. The “history” part of the name comes from the fact that block producers hash transaction id’s into the stream to record which transactions were processed in their block.

PoH hash calculation :next_hash = hash(prev_hash, hash(transaction_ids))

PoH can be used as a trusted clock because each hash must be produced sequentially. Each produced block contains a blockhash and a list of hash checkpoints called “ticks” so that validators can verify the full chain of hashes in parallel and prove that some amount of time has actually passed.

Transaction Expiration#

By default, all Solana transactions will expire if not committed to a block in a certain amount of time. Thevast majority of

transaction confirmation issues are related to how RPC nodes and validators detect and handleexpired transactions. A solid understanding of how transaction expiration works should help you diagnose the bulk of your transaction confirmation issues.

How does transaction expiration work?#

Each transaction includes a “recent blockhash” which is used as a PoH clock timestamp and expires when that blockhash is no longer “recent enough”.

As each block is finalized (i.e. the maximum tick heightis reached , reaching the "block boundary"), the final hash of the block is added to theBlockhashQueue which stores a maximum of the300 most recent blockhashes . During transaction processing, Solana Validators will check if each transaction's recent blockhash is recorded within the most recent 151 stored hashes (aka "max processing age"). If the transaction's recent blockhash isolder than this max processing age, the transaction is not processed.

Info Due to the currentmax processing age of 150 and the "age" of a blockhash in the queue being0-indexed , there are actually 151 blockhashes that are considered "recent enough" and valid for processing. Sinceslots (aka the time period a validator can produce a block) are configured to last about400ms , but may fluctuate between 400ms and 600ms, a given blockhash can only be used by transactions for about 60 to 90 seconds before it will be considered expired by the runtime.

Example of transaction expiration#

Let’s walk through a quick example:

1. A validator is actively producing a new block for the current slot
2. The validator receives a transaction from a user with the recent blockhashabcd...
3. The validator checks this blockhashabcd...
4. against the list of recent
5. blockhashes in theBlockhashQueue
6. and discovers that it was created 151
7. blocks ago
8. Since it is exactly 151 blockhashes old, the transaction has not expired yet
9. and can still be processed!
10. But wait: before actually processing the transaction, the validator finished
11. creating the next block and added it to theBlockhashQueue
12. . The validator
13. then starts producing the block for the next slot (validators get to produce
14. blocks for 4 consecutive slots)
15. The validator checks that same transaction again and finds it is now 152
16. blockhashes old and rejects it because it’s too old :(

Why do transactions expire?#

There’s a very good reason for this actually, it’s to help validators avoid processing the same transaction twice.

A naive brute force approach to prevent double processing could be to check every new transaction against the blockchain’s entire transaction history. But by having transactions expire after a short amount of time, validators only need to check if a new transaction is in a relatively small set ofrecently processed transactions.

Other blockchains#

Solana’s approach of prevent double processing is quite different from other blockchains. For example, Ethereum tracks a counter (nonce) for each transaction sender and will only process transactions that use the next valid nonce.

Ethereum’s approach is simple for validators to implement, but it can be problematic for users. Many people have encountered situations when their Ethereum transactions got stuck in apending state for a long time and all the later transactions, which used higher nonce values, were blocked from processing.

Advantages on Solana#

There are a few advantages to Solana’s approach:

1. A single fee payer can submit multiple transactions at the same time that are
2. allowed to be processed in any order. This might happen if you’re using
3. multiple applications at the same time.
4. If a transaction doesn’t get committed to a block and expires, users can try
5. again knowing that their previous transaction will NOT ever be processed.

By not using counters, the Solana wallet experience may be easier for users to understand because they can get to

success, failure, or expiration states quickly and avoid annoying pending states.

Disadvantages on Solana#

Of course there are some disadvantages too:

1. Validators have to actively track a set of all processed transaction id's to
2. prevent double processing.
3. If the expiration time period is too short, users might not be able to submit
4. their transaction before it expires.

These disadvantages highlight a tradeoff in how transaction expiration is configured. If the expiration time of a transaction is increased, validators need to use more memory to track more transactions. If expiration time is decreased, users don't have enough time to submit their transaction.

Currently, Solana clusters require that transactions use blockhashes that are no more than 151 blocks old.

Info ThisGithub issue contains some calculations that estimate that mainnet-beta validators need about 150MB of memory to track transactions. This could be slimmed down in the future if necessary without decreasing expiration time as are detailed in that issue.

Transaction confirmation tips#

As mentioned before, blockhashes expire after a time period of only 151 blocks which can pass as quickly asone minute when slots are processed within the target time of 400ms.

One minute is not a lot of time considering that a client needs to fetch a recent blockhash, wait for the user to sign, and finally hope that the broadcasted transaction reaches a leader that is willing to accept it. Let's go through some tips to help avoid confirmation failures due to transaction expiration!

Fetch blockhashes with the appropriate commitment level#

Given the short expiration time frame, it's imperative that clients and applications help users create transactions with a blockhash that is as recent as possible.

When fetching blockhashes, the current recommended RPC API is calledgetLatestBlockhash . By default, this API uses thefinalized commitment level to return the most recently finalized block's blockhash. However, you can override this behavior bysetting thecommitment parameter to a different commitment level.

Recommendation

Theconfirmed commitment level should almost always be used for RPC requests because it's usually only a few slots behind theprocessed commitment and has a very low chance of belonging to a droppedfork .

But feel free to consider the other options:

- Choosingprocessed
- will let you fetch the most recent blockhash compared to
- other commitment levels and therefore gives you the most time to prepare and
- process a transaction. But due to the prevalence of forking in the Solana
- blockchain, roughly 5% of blocks don't end up being finalized by the cluster
- so there's a real chance that your transaction uses a blockhash that belongs
- to a dropped fork. Transactions that use blockhashes for abandoned blocks
- won't ever be considered recent by any blocks that are in the finalized
- blockchain.
- Using thedefault commitment
- levelfinalized
- will eliminate any risk that the blockhash you choose will belong to a dropped
- fork. The tradeoff is that there is typically at least a 32 slot difference
- between the most recent confirmed block and the most recent finalized block.
- This tradeoff is pretty severe and effectively reduces the expiration of your
- transactions by about 13 seconds but this could be even more during unstable
- cluster conditions.

Use an appropriate preflight commitment level#

If your transaction uses a blockhash that was fetched from one RPC node then you send, or simulate, that transaction with a different RPC node, you could run into issues due to one node lagging behind the other.

When RPC nodes receive asendTransaction request, they will attempt to determine the expiration block of your transaction using the most recent finalized block or with the block selected by thepreflightCommitment parameter. AVERY common issue is that a received transaction's blockhash was produced after the block used to calculate the expiration for that transaction. If an RPC node can't determine when your transaction expires, it will only forward your transactionone time and afterwards will thendrop the transaction.

Similarly, when RPC nodes receive asimulateTransaction request, they will simulate your transaction using the most recent finalized block or with the block selected by thepreflightCommitment parameter. If the block chosen for simulation is older than the block used for your transaction's blockhash, the simulation will fail with the dreaded "blockhash not found" error.

Recommendation

Even if you useskipPreflight ,ALWAYS set thepreflightCommitment parameter to the same commitment level used to fetch your transaction's blockhash for bothsendTransaction andsimulateTransaction requests.

Be wary of lagging RPC nodes when sending transactions#

When your application uses an RPC pool service or when the RPC endpoint differs between creating a transaction and sending a transaction, you need to be wary of situations where one RPC node is lagging behind the other. For example, if you fetch a transaction blockhash from one RPC node then you send that transaction to a second RPC node for forwarding or simulation, the second RPC node might be lagging behind the first.

Recommendation

ForsendTransaction requests, clients should keep resending a transaction to a RPC node on a frequent interval so that if an RPC node is slightly lagging behind the cluster, it will eventually catch up and detect your transaction's expiration properly.

ForsimulateTransaction requests, clients should use thereplaceRecentBlockhash parameter to tell the RPC node to replace the simulated transaction's blockhash with a blockhash that will always be valid for simulation.

Avoid reusing stale blockhashes#

Even if your application has fetched a very recent blockhash, be sure that you're not reusing that blockhash in transactions for too long. The ideal scenario is that a recent blockhash is fetched right before a user signs their transaction.

Recommendation for applications

Poll for new recent blockhashes on a frequent basis to ensure that whenever a user triggers an action that creates a transaction, your application already has a fresh blockhash that's ready to go.

Recommendation for wallets

Poll for new recent blockhashes on a frequent basis and replace a transaction's recent blockhash right before they sign the transaction to ensure the blockhash is as fresh as possible.

Use healthy RPC nodes when fetching blockhashes#

By fetching the latest blockhash with theconfirmed commitment level from an RPC node, it's going to respond with the blockhash for the latest confirmed block that it's aware of. Solana's block propagation protocol prioritizes sending blocks to staked nodes so RPC nodes naturally lag about a block behind the rest of the cluster. They also have to do more work to handle application requests and can lag a lot more under heavy user traffic.

Lagging RPC nodes can therefore respond togetLatestBlockhash requests with blockhashes that were confirmed by the cluster quite awhile ago. By default, a lagging RPC node detects that it is more than 150 slots behind the cluster will stop responding to requests, but just before hitting that threshold they can still return a blockhash that is just about to expire.

Recommendation

Monitor the health of your RPC nodes to ensure that they have an up-to-date view of the cluster state with one of the following methods:

1. Fetch your RPC node's highest processed slot by using thegetSlot
2. RPC API with theprocessed
3. commitment level and then call the`getMaxShredInsertSlot
4. RPC API to
5. get the highest slot that your RPC node has received a "shred" of a block
6. for. If the difference between these responses is very large, the cluster is
7. producing blocks far ahead of what the RPC node has processed.
8. Call thegetLatestBlockhash
9. RPC API with theconfirmed

10. commitment level
11. on a few different RPC API nodes and use the blockhash from the node that
12. returns the highest slot for itscontext slot
13. .

Wait long enough for expiration#

Recommendation

When calling thegetLatestBlockhash RPC API to get a recent blockhash for your transaction, take note of thelastValidBlockHeight in the response.

Then, poll thegetBlockHeight RPC API with theconfirmed commitment level until it returns a block height greater than the previously returned last valid block height.

Consider using “durable” transactions#

Sometimes transaction expiration issues are really hard to avoid (e.g. offline signing, cluster instability). If the previous tips are still not sufficient for your use-case, you can switch to using durable transactions (they just require a bit of setup).

To start using durable transactions, a user first needs to submit a transaction thatinvokes instructions that create a special on-chain “nonce” account and stores a “durable blockhash” inside of it. At any point in the future (as long as the nonce account hasn’t been used yet), the user can create a durable transaction by following these 2 rules:

1. The instruction list must start with an“advance nonce” system instruction
2. which loads their on-chain nonce account
3. The transaction’s blockhash must be equal to the durable blockhash stored by
4. the on-chain nonce account

Here’s how these durable transactions are processed by the Solana runtime:

1. If the transaction’s blockhash is no longer “recent”, the runtime checks if
2. the transaction’s instruction list begins with an “advance nonce” system
3. instruction
4. If so, it then loads the nonce account specified by the “advance nonce”
5. instruction
6. Then it checks that the stored durable blockhash matches the transaction’s
7. blockhash
8. Lastly it makes sure to advance the nonce account’s stored blockhash to the
9. latest recent blockhash to ensure that the same transaction can never be
10. processed again

For more details about how these durable transactions work, you can read theoriginal proposal andcheck out an example in the Solana docs.

Previous «Address Lookup Tables Next Retrying Transactions»