SpaceXpanse ROD blockchain specs

This repository contains technical specifications and design documents that describe how the various components and layers in the ecosystem interact. In particular, important topics are:

- SpaceXpanse ROD blockchain *tech specs*
- The basic SpaceXpanse ROD blockchain consensus protocol
- SpaceXpanse ROD blockchain *triple-purpose mining* that secures the blockchain
- A standard for currencies on the SpaceXpanse Multiverse platform
- How games should interact with the core blockchain
- In-game trading of assets for ROD coin using atomic name updates
- The core daemon interface for game engines

You can access SpaceXpanse Multiverse documentation here >
https://github.com/SpaceXpanse/Documentation/wiki

Content

SpaceXpanse ROD coin tech specs

Name: SpaceXpanse
Ticker: ROD
Symbol: Ɍ
Address letter: R
Address header: 60
P2SH header: 75
Port: 11998
RPC port: 11999
Algorithm: **neoscrypt-xaya** and **SHA-256d**, for merged mining - ChainID: 1899
Block reward: **800 ROD**[1], halving every 1,054,080 blocks /~1 year/ for **5 years** and then imposing **~3% inflation** on the supply of that exact moment until **59 years** are passed and the supply reaches ~2.5 fold increment.
Reward distribution: **75%/25%** - 3 neoscrypt-xaya blocks and 1 SHA-256d block every 4 blocks till the beginning of the **third year** when **masternodes** with delegated proof of stake /DPoS/ will be introduced /TBD/.
Block time: 30 seconds
Spend: 6 confirmations
Block: 120 confirmations
Current supply: ~700,000,000 ROD
Max. supply: 4,615,066,365[2] ROD
ICO/IEO: None
Premine: 199,999,998 coins in the developer's multisig 3/4 address and 2/3 of them will be transferred to time-locked addresses to be unlocked in the next four years[3]. The other will be used to support the first year of development.

---

[1]There is a pre-release period of 55560 blocks /around 01.07.2020 11:59 PM UTC/ in the beginning, in which the reward will be 1 ROD.
[2]They will be unspendable until target block numbers are reached. The unused coins from the premine will be burned accordingly.

Blockchain consensus protocol

This document gives an overview of the low-level consensus protocol implemented by the core SpaceXpanse daemon.

NOTE: The final reference for the protocol is the SpaceXpanse Core implementation. This document just highlights the most important parts in an easy-to-read form.

SpaceXpanse is based on Namecoin

Most parts of the SpaceXpanse consensus protocol are inherited from Namecoin, which itself inherits most of the Bitcoin protocol.

As with Bitcoin, Namecoin and SpaceXpanse implement a distributed ledger by tracking the current set of "unspent transaction outputs". However, in addition to pure currency transactions, they also implement a name-value database. This database is stored and updated similar to the UTXO set. Each entry contains the following fields:

- Name: A byte array with restrictions that is the "key" for the name-value-database. In SpaceXpanse, this is typically the account name of a player. It can be up to 256 bytes.
- Value: Another, typically longer, byte array (also with restrictions) that holds data associated with the name. In SpaceXpanse, this is used to store a player's latest moves or other actions. It can be up to 2,048 bytes.
- Address: Similar to transaction outputs, each name is associated with a SpaceXpanse address (Bitcoin script) that "owns" it. Only the owner has *write access* to this name's entry, while everyone can *read* it from the database. The owner can send the name to a different address, either one of their own or to someone else's.

In addition to spending and creating currency outputs, each transaction can optionally also register or update a name in the database. A single transaction can at most affect one name. A name can be updated at most once per block.

Specific details for the transaction format are not provided here because they are the same as in Namecoin.

In contrast to Namecoin, which registers names in a two-step process (`name_new` and `name_firstupdate`), SpaceXpanse's name registrations are always done in a single transaction (called `name_register`). This transaction is similar to a `name_update` in Namecoin except that it does not consume a name input.

Basic differences from Bitcoin

Some of the basic chain parameters and properties of the genesis block in SpaceXpanse differ from those in Bitcoin and Namecoin. Furthermore, some of these details were changed in the past with scheduled forks. The current parameters for `mainnet` are:

- PoW mining is done based on triple-purpose mining.
- The difficulty is updated for each block using the Dark Gravity Wave (DGW) formula.
- Blocks are scheduled to be produced, on average, every 30 seconds instead of every 10 minutes. Difficulty retargets independently for each of the two possible mining algorithms. Merge mined SHA-256d blocks are scheduled once every two minutes and standalone Neoscrypt blocks every 40 seconds.
- The initial block reward is set to 800 ROD, and halves every 1054080 blocks, which corresponds to Bitcoin's halving of once a year.
- The genesis block's coinbase transaction pays to a multisig address owned by the SpaceXpanse team. Unlike Bitcoin and Namecoin, it is actually spendable, and does not observe the usual "block maturity" rule.
    - These coins will be used to support the project development. They will be equally distributed and sent to time-locked addresses which will unlock every 1054080 block (every year) for the next 3 years. Unspent coins will be destroyed by sending them to a provably unspendable script (`OP_RETURN`).
- The maximum block weight is 400,000 instead of 4 million, corresponding to a block size of 100 KB instead of 1 MB. The maximum number of sigops in a block is 8,000.
- The maximum size of a script element is 2,048 bytes instead of 520 bytes, primarily to allow for larger name values.

Activation of soft forks

SpaceXpanse immediately activates some of the soft forks introduced in Bitcoin over time since we start with a fresh blockchain:

- BIP 16
- BIP 34

- BIP 65
- BIP 66
- CSV (BIP 68, BIP 112 and BIP 113)
- Segwit (BIP 141, BIP 143 and BIP 147)

(Note that some of these activate later or never on the regtest network to allow for certain tests to be possible.)

Name and value restrictions

Valid names and values in SpaceXpanse must satisfy additional constraints compared to Namecoin, which only enforces maximum lengths. In particular, these conditions must be met by names and values:

- Names can be up to 256 bytes long and values up to 2,048 bytes.
- Names must have a namespace. This means that they must start with one or more lower-case letters and `/`. Expressed as a regexp, this means they must match: `[a-z]+\/.*`
- Names must be valid UTF-8 and must not contain unprintable ASCII characters, i.e. those with a value less than 0x20.
- Values must be valid JSON and parse to a JSON object.

SpaceXpanse ROD triple-purpose mining

SpaceXpanse is committed to proof-of-work (PoW) for securing its blockchain. As there are various drawbacks to commonly-used PoW schemes, SpaceXpanse implements a new design that unifies the best of all worlds.

The following provides a high-level design overview and then describes the technical details of the SpaceXpanse triple-purpose mining implementation.

Common PoW schemes and their drawbacks

Blockchain projects that use PoW typically choose from one of two common schemes: Merged mining with a more prominent chain (e.g. Bitcoin), or if *not* merge-mining, a combination of one or more "ASIC resistant" hash algorithms. Both of these choices have their unique advantages and drawbacks.

Merged mining

Merged mining allows existing miners of a PoW blockchain (e.g. Bitcoin) to mine a second blockchain at the same time and almost for free. This allows multiple blockchains to share hashing power, securing all of them instead of reducing the security of each chain by splintering the mining community.

As a result, merge-mined blockchains have a very high hash rate and are more resistant to 51% attacks. For instance, Namecoin (the blockchain that pioneered merged mining) had at one point in time a hash rate even higher than Bitcoin. This happened because the existing pool of Bitcoin miners split between mining Bitcoin and Bitcoin Cash, while miners of both chains could merge-mine Namecoin.

On the other hand, if at least some of the big Bitcoin mining pools decide to merge-mine a new blockchain, they will almost surely control by orders of magnitude more hash power than any individual miner could contribute. (This is also what makes merged mining secure!) However, this also means that the aspect of distributing coins to a wider community, which is also a goal of mining, is lost. Further, sudden large drops in the hash rate are possible when one of the pools ceases merged mining, which can slow the blockchain down considerably.

Stand-alone mining

If a blockchain decides against merged mining, this means that each and every miner for this blockchain must be convinced to dedicate their hash power to the new chain exclusively. This typically means that, especially in the beginning of a new project, only smaller miners join.

While that is beneficial for decentralization of mining and to distribute coins widely, it invariably means that the total hash rate is very low and thus that the chain is more susceptible to 51% attacks. Even if an "ASIC resistant" mining algorithm is chosen (such that existing Bitcoin mining chips cannot be used), it's still relatively cheap for an attacker to gain a large hash rate for a short time by renting GPUs or CPUs from a cloud provider.

High-level overview for triple-purpose mining

To combine the best of both worlds (merged and stand-alone mining), we propose triple-purpose mining as a new scheme that forms a good compromise between the two extremes:

SpaceXpanse blocks can be either merge-mined with SHA-256d (Bitcoin), or they can be mined stand-alone with a modified Neoscrypt. The *chain ID* for merge mining SpaceXpanse is `1899`. There are no particular rules enforcing a certain sequence of blocks for each algorithm (unlike some existing multi-algorithm projects), but the difficulty for each algorithm is retargeted independently. This means that, on average and independent of the distribution of hash rate between the two algorithms, there is one SHA-256d and one Neoscrypt block every minute (leading to an average rate of one block every 30 seconds). Furthermore, block rewards are not equal; instead, 75% of the total PoW coin supply goes to stand-alone Neoscrypt blocks, while 25% goes to merge-mined SHA-256d blocks.

The three main benefits ("purposes") of this are:

1. Due to merge mining, the total hash rate is very high and the chain is thus highly resistant to 51% attacks. (Note that, as in Bitcoin, it is not the actual *length* of a chain that counts, but the *work* it contains. This means that even if the Neoscrypt hash rate is very low and a single miner controls almost 100% of it, then they will likely still have much less than 50% of the total hash rate and thus not be able to run a 51% attack.)
2. Stand-alone mining with Neoscrypt and a relatively high block reward for these blocks makes it possible to distribute coins to a wide community of individual

miners. However, since merge mining is essentially free, there is still sufficient incentive for mining pools to merge-mine even with only 25% of total PoW coins going to them.

3. By producing one block per minute individually from both mining algorithms, we greatly increase the resilience of the blockchain against stalling when one of the algorithms has a sharp drop in hash rate. (Even if *all* mining of one algorithm were to stop temporarily, the blockchain would still continue to produce blocks on average once per minute.)

Technical details

Huntercoin already implements dual-algorithm mining (although allowing both algorithms to be merge-mined). However, the exact scheme used for merge-mining there (inherited from Namecoin) is not ideal. Since it uses the `nVersion` field of the block header to signal merge mining (and, in the case of Huntercoin, which algorithm is used), it conflicts with BIP 9. Thus, merge mining in SpaceXpanse is implemented differently and does not use the `nVersion` field of the main block header.

PoW data in the block header

For PoW in SpaceXpanse, the hash of the actual block header never matters. Instead, each block header is always followed by special PoW data. This contains metadata about the PoW (the algorithm used and whether or not it was merge-mined) as well as the actual data proving the work by committing to the SHA-256d hash of the actual block header.

The first byte of the PoW data indicates the mining algorithm:

Value	Algorithm
`0x01`	SHA-256d
`0x02`	Neoscrypt

In addition, when the block is merge-mined, the highest-value bit (`0x80`) is also set to indicate this. This means that the only valid values are `0x81` (merge-mined SHA-256d) and `0x02` (stand-alone Neoscrypt).

The difficulty target for the chosen algorithm follows in the `nBits` field. *The `nBits` field in the block header is unused, and must be set to zero.* (Without the PoW data, the block header alone does not specify the mining algorithm, so it doesn't make sense to specify the difficulty in it.) When validating a block header with the PoW data, the usual rules apply:

- The block header `nBits` field must be zero.
- The PoW data `nBits` field must match the expected difficulty for the selected algorithm, following the difficulty retargeting for that algorithm.
- The PoW must match the `nBits` difficulty target specified in the PoW data.

The format of the remainder of the PoW data depends on whether or not merged mining is used. If it is, then an "auxpow" data structure as per Namecoin's merged mining follows. If the block is stand-alone mined, then 80 bytes follow, such that:

1. Their hash according to the selected algorithm (Neoscrypt) satisfies the difficulty target.
2. Bytes 37 to 68, where the Merkle root hash would be in a Bitcoin block header, contain exactly the hash of the SpaceXpanse block header.

Apart from the block hash being in the specified position, there are no other requirements for these bytes. They can set other fields similar to how a block header would set them, but are not required to.

However, note that such a proof can *never* be used also as auxpow itself, since it has to contain the block hash in the position of the Merkle root. This makes it impossible (barring a SHA-256d collision) to connect a coinbase transaction to it, which would be required for a valid auxpow. This would prevent using a single proof for two blocks (as stand-alone and auxpow), even if it were possible to use a single algorithm both for merged and stand-alone mining.

This particular format for attaching PoW to block headers has various benefits:

- It does not put *any* constraints at all on the actual block header, which prevents conflicts with BIP 9 as well as similar problems in the future.
- It reuses the existing format for merged mining as far as possible, and, in particular, allows merge-mining SpaceXpanse together with existing chains as Namecoin does.
- The data that is hashed for stand-alone mining has the same format as a Bitcoin block header, so that existing software and mining infrastructure built for Bitcoin-like blocks can be used.

- Instead of the actual block header, we always feed derived data committing to its hash to the mining application. This makes it easier to add more data to the block header in a future hard fork if desired (e.g. for implementing Ephemeral Timestamps).

Mining interfaces of the core daemon

The core daemon provides different RPC methods that can be used by external mining infrastructure:

`getblocktemplate`

The `getblocktemplate` RPC method is available as in upstream Bitcoin and can be used by advanced users. This requires the external miner to construct the full resulting block, including the correct PoW data, themselves. (This requirement is similar for miners using `getblocktemplate` with existing merge-mined coins like Namecoin.)

`createauxblock` and `submitauxblock`

For merge-mining SpaceXpanse, the RPC methods `createauxblock` and `submitauxblock` are provided similar to Namecoin. They handle the construction of the block with the correct format, as long as the miner can construct the auxpow itself (as is required for Namecoin).

`getwork`

For out-of-the-box stand-alone mining, SpaceXpanse provides the `getwork` RPC method that was previously used in Bitcoin. It constructs the PoW data as described above internally and returns the "fake block header" data that needs to be hashed, such that existing mining tools can readily process it.

Integration with stratum

Stratum is a widely used protocol for mining hardware, software and pools. It defines a way for miners to receive and submit work.

Instead of just receiving the data to hash, they receive the fields necessary to construct the final block header and coinbase transaction, and just submit their nonces back. This allows for efficient data transfer and also gives miners the power to roll an extra nonce value (that is normally part of the coinbase transaction) and thus increase the search space before they have to communicate again with the server.

It is possible to "fit" SpaceXpanse's stand-alone mining format into the Stratum protocol (even though SpaceXpanse modifies the format of block headers with the PoW data). With the following changes done *server side*, Stratum clients can just work out of the box (if they implement SpaceXpanse's Neoscrypt variant as hashing algorithm):

In Stratum, the miner themselves builds up the final coinbase transaction from a template, hashes it to compute the Merkle root, and then uses this to construct the actual block header. It then computes PoW on that constructed block header. To make this work with SpaceXpanse's PoW data and the "fake block header", the Stratum client receives the real SpaceXpanse block header as "coinbase template", together with an empty Merkle branch. The client will then put its hash into the "Merkle root" of the fake block header and solve PoW on it.

The real block header's `nNonce` field can be chosen arbitrarily. We use these four bytes as the place where the Stratum miner will place the "extra nonce 1" and "extra nonce 2" when constructing the final "coinbase" (which is the real block header in SpaceXpanse). We can use two bytes for each of them.

Thus, the full server-side Stratum implementation for SpaceXpanse works like this:

- When a client requests work, the Stratum server sends the real SpaceXpanse block header except the `nNonce` field as "coinbase part 1", an empty string as "coinbase part 2" and an empty Merkle branch. `nTime` and `nBits` are sent as usual; `nVersion` and the previous block header (which the client will use inside the fake block header) can be chosen e.g. as all zeros.
- The client constructs what it believes to be the "coinbase", which in fact will be the completed SpaceXpanse block header. It hashes it with SHA-256d and puts the hash into the "Merkle root" field of the fake block header (which it believes to be the actual block header), as required.
- When the client has found a valid PoW for the fake block header, it submits the nonces for it back to the server.
- The server can then use all the data to build the actual SpaceXpanse block, including the real block header with the client's extra nonces, and the PoW data based on the other fields and the client's nonce.

SpaceXpanse ROD blockchain currencies

Currencies (custom tokens issued on the SpaceXpanse blockchain) are a special case of more general games, and can thus be implemented and managed through a SpaceXpanse game state.

This document defines a standard for how such "games" may be created with certain parameters, what the game rules are for them and what moves can be made in the game to transact the tokens. All currencies following this standard can then be supported out-of-the-box by wallets or other infrastructure. (This is very similar to how ERC-20 standardized tokens on Ethereum.)

Creation of a new currency

To create a new currency, a corresponding `g/` name must be registered. In the registration, the name's initial value must specify the new game to be a currency according to this standard and set the basic parameters. Wallets or other tools that wish to support general currencies must watch the blockchain for registrations of `g/` names specifying valid parameters to include them in their system. (Or at least list them as available currencies and give the user a choice to enable them.)

All name updates other than the initial registration are ignored, so the parameters must be set during registration of the name and cannot be changed afterwards anymore!

The initial value for a currency `g/` name must be a JSON object of the following form:

```
{
  "type": "currency",
  "version": 1,
  "creator": CREATOR,
  "supply": SUPPLY,
  "fixed": FIXED,
}
```

The `version` field must currently be set to `1` to indicate that the currency follows the first version of this standard. In the future, additional versions may be defined. The meanings of the placeholders are:

- `CREATOR`: The player account (without the `p/` prefix) who is considered creator of the currency.

- `SUPPLY`: The total (initial) supply of tokens for this currency (must be an integer). They are initially all assigned to the currency's creator.
- `FIXED`: Boolean value that indicates whether the token supply is fixed with the creation or not. If it is not fixed, then the creator is allowed to issue new tokens (see below).

For example, this creates a currency with a fixed supply of 1,000 tokens that are all initially assigned to `p/alice`:

```
{
  "type": "currency",
  "version": 1,
  "creator": "alice",
  "supply": 1000,
  "fixed": true
}
```

Game state of a currency

The current state of a currency (its "game state") is fully defined by a map of player names to the amount of tokens they hold.

All token amounts are integers. They are interpreted to be the 10e8-th part of the token's basic unit, similar to satoshis and bitcoins. Wallets and other user interfaces must convert between the integer amounts and the displayed "token" amounts.

Transactions

Moves referencing the currency's game ID are used to *transfer tokens to someone else*. The sender of the transaction (from whose balance the amount is deducted) is the account that sent the move by updating its `p/` name. Tokens can also be burnt (provably destroyed). If the token supply is not fixed (as specified by the game's parameters), then the creator of the currency can also issue new tokens through a move.

The JSON value defining a currency transaction looks like this:

```
{
  "s":
    {
      RECIPIENT1: AMOUNT1,
      RECIPIENT2: AMOUNT2,
      ...
    },
  "b": BURN-AMOUNT,
  "c": CREATE-AMOUNT,
}
```

Each of these three fields is optional. The placeholders' meanings are:

- `RECIPIENT`n: The player name (without `p/` prefix) to whom tokens are sent.
- `AMOUNT`n: The amount of tokens (integer) to send to `RECIPIENT`n.
- `BURN-AMOUNT`: The amount of tokens to burn (destroy irrecoverably and provably).
- `CREATE-AMOUNT`: The amount of tokens to create. They are added to the balance of the user sending the move.

A move is only valid if *both* of the following conditions are fulfilled:

1. If `c` is set, then the game must be specified with `fixed: false` and the user sending the move must be the game's creator.
2. The user's balance must be large enough. More specifically, the value of `AMOUNT1 + AMOUNT2 + ... + BURN-AMOUNT` must be less than or equal to the user's current balance (per the game state) plus `CREATE-AMOUNT`.

If at least one of these conditions is not true, then the whole move is invalid and does not affect the game state *at all*!

For example, if a currency `g/gold` is defined and `p/bob` has at least ten gold units, then the following update of the name `p/bob` is a valid transaction. It burns two gold units, sends five units to `p/alice` and three to `p/charly`:

```
{
  "g":
    {
      "gold":
        {
          "b": 200000000,
          "s":
            {
              "alice": 500000000,
              "charly": 300000000
            }
        }
    }
}
```

Trading and atomic transactions

For a currency as defined here, it is very useful to have the opportunity to trade it against ROD or another currency, and to do atomic transactions with trading partners.

Note that it is not possible to do atomic transactions with currencies as defined here in a straight-forward way (i.e. by sending them between trading partners in a single transaction). There are two reasons for this:

1. A transaction as specified here is done through a name update of the *sending name*. Since each SpaceXpanse transaction can contain at most one name operation, this means that it is only possible for someone to send multiple currencies in a single transaction, but not for *two different* people to transfer to each other as required for atomic trading.
2. As described above, even if a name update for sending tokens makes it into the blockchain it is *not guaranteed that the tokens are actually sent*. The transaction might be invalid and ignored if the *sender's balance is not sufficient*. For an atomic transaction, it would have to be ensured that either both sends are executed or none is; and for that to be possible, the update to one currency's ledger would have to depend on the state of the other, linking the two game states. This is not desired, because we want to give users the ability to track only currencies they care about.

Temporary names ("Vessels")

Instead, currencies can be traded for ROD atomically by using temporary *vessel* names: If Alice wants to sell 100 of some token for 100 ROD to Bob, she creates a temporary name in her wallet and sends the tokens to it. Then, Alice and Bob do an *atomic name trade*, where Bob buys the vessel name from Alice for 100 ROD. Finally, Bob can transfer the tokens out of the vessel (or keep them there).

Note that if Alice sends the tokens out of the vessel before the name trade is confirmed, this *double spends the vessel name* and thus also *invalidates the whole trading transaction*. In other words, Alice also won't get the ROD, so that these kinds of transactions are safe.

Vessel names in a user's wallet can be reused, so that there is not an ever-growing number of unused names cluttering the blockchain. Since each name registration costs the locked amount of 0.01 ROD, there is incentive for users to reuse vessels where possible.

Implementation notes

Wallets and other software handling currencies are effectively "game engines". As such, they must not only handle moves from new blocks, but they must also be able to undo blocks that are detached.

Transactions following the specification above can be undone easily, since they already contain all the information necessary to restore the previous state:

- Sent amounts are deducted from the recipients' balances and added to the balance of the sender of the move.
- Burnt amounts are added to the balance of the sender.
- Created amounts are deducted from the sender's balance.

However, game engines must know whether or not a given transaction was actually valid, and only undo the valid ones! Otherwise, they might "undo" the sending of a large amount that never belonged to the player that sent the move.

Thus, the undo data that implementations should keep for every block is the list of transactions that were either valid or invalid. (Which one does not matter and might be chosen depending on which list is shorter. That will typically be the list of invalid moves, but it can also be the other way round especially if a deliberate DoS attack is attempted.) This list should ideally be stored using transaction IDs, although it is also possible to use the list of sending names (since each name can only be updated in a single transaction for each block).

There may be further optimisations possible to reduce the storage requirement, but since even the full list of transaction IDs is already much smaller than the block data itself, any larger effort is unlikely to be worthwhile.

Games and other complex applications on ROD blockchain

This document describes how games in SpaceXpanse are structured and how they interact with the core SpaceXpanse blockchain. By following these rules, games ensure that they fit well into the SpaceXpanse ecosystem and that the game-specific SpaceXpanse interface works well for them.

Basic model

Each game has a game state, which represents, for instance, the current state and position of each avatar, in-game assets that each player owns, etc. A particular state is associated with each block in the SpaceXpanse blockchain, and all nodes running a particular game have consensus about the current state. This state is roughly speaking what the UTXO set is in Bitcoin, except that it may be more complex as needed for the particular game.

NOTE: Each *block hash* has a unique state associated with it; but if a chain reorg occurs, then different blocks at the same *height* in the chain will typically have *different* game states associated with them. So when storing or processing game states, the block hash and not the block height should be used as the unique key!

The state is then modified from one block to the next by processing all moves done by players in the new block; this is similar to how Bitcoin transactions in each block update the UTXO set when attaching a new block to the Bitcoin chain. How exactly the game state is updated depends on the game rules, and this is the core functionality that each game engine must implement. In other words, the game engine needs a function that maps an old state and a list of moves to a new state:

`f`: (old state, moves) -> new state

This function processes the game state forward in time. In addition, to properly handle chain reorgs, there must be a way to *restore old game states*. This is discussed below.

In general, it is the responsibility of the game engine to keep track of the current state and its updates. The SpaceXpanse daemon, on the other hand, provides the game engine with information about each attached or detached block and the moves performed in them.

Moves

Player accounts are represented in SpaceXpanse by names with the `p/` prefix. For instance, `p/jamesholden` is for the player account "jamesholden". Each player performs moves by updating this name to a value that encodes the move(s) he wants to make.

The updated name value must be a JSON object. If it contains moves associated with a particular game (which is identified by a unique game id), then the move data should be stored as a JSON value in `.g[GAMEID]`. The game decides on the type and format of the move data, as well as how it processes the move when updating the game state.

As a fundamental rule, game engines should only ever depend on SpaceXpanse transactions that are name updates (or registrations) referencing the particular game ID! This can be `p/` names that store a move in `g[GAMEID]`, and it can be updates to the `g/` name itself. The game state must not depend on any other transactions, neither pure currency transactions nor name updates not referencing the game. Furthermore, the game engine must only take into account the actual move value from the name update, not any other data. It may, however, also depend on *currency outputs* created in the same transaction and the *spent inputs* (useful for atomic trades).

In particular, games must also not include any newly registered names into the game state until those names have been referenced by a move associated with the game's ID. Typically, players should make an explicit move with a new name that indicates they wish to join the game or create an avatar in the game. (Of course, game UIs may show a list of all the names the player owns so that the player can easily choose to join in this way through the UI.)

Here are some example values:

- `{}`: This is the minimal valid JSON object, and can be used as a value when a name is registered or sent to a different address if no moves are intended to be made at the same time. Name updates with this value are not sent to any game engine.
- `{"g":{"chess":"e8Q"}}`: A move (encoded as string) for the game `chess`.

More complicated examples may also contain moves encoded as JSON objects, other fields on the top level (they are not yet specified and will be ignored), and moves in two games at the same time:

```
{
  "chat": "SpaceXpanse rocks!",
  "g":
    {
      "chess": "0-0-0",
      "huc":
        {
          "wp": [0, 0, 42, 48, 0, 0]
        }
    }
}
```

- If a name is updated to this value, then both the `chess` and `huc` game engines are notified (if installed). The `chess` engine must only process the chess move `"0-0-0"`, though, and must ignore all other parts. Similarly, the `huc` engine must only process the JSON object for the `huc` move.

Game IDs

Since the move format references particular games, there must be unique IDs for each game. These game IDs are strings, and game creators must reserve them by registering the name `g/GAMEID` on the SpaceXpanse blockchain. This ensures uniqueness.

NOTE: This rule is not strictly enforceable by the blockchain, but it is in the game creator's own interest to follow this guideline.

Ownership of the name corresponding to a game can be used to prove to every node that someone is the "owner" of a game. This can be used to send update notifications or other trusted communication to all users of a particular game. In particular, `g/` names of a game can send *admin commands* to all instances of the game. For that, the name should be updated to a value that contains a custom JSON value (encoding the command in a game-specific form) in the `cmd` field of the top-level JSON object. For instance:

```
{
  "cmd":
    {
      "type": "setparam",
      "name": "goldprice",
      "value": 100,
    },
  "other stuff": "ignored",
}
```

Duplicate JSON keys

The JSON standard allows object values to contain duplicate keys. Ideally, this "feature" should not be used / relied upon at all, especially since many JSON libraries cannot handle this very well.

Moves

From the point of view of moves in a SpaceXpanse game, these situations with duplicated JSON keys are interesting:

```
{
  "g": {"first": "move 1"},
  "g": {"second": "move 2"},
}
```

For a duplicated `g` field like this, SpaceXpanse will process all of them in order. In the example above, the transaction specifies two valid moves, one for `g/first` and the other for `g/second`.

```
{
  "g":
    {
      "my game": "move a",
      "my game": "move b"
    }
}
```

If the game ID itself is duplicated (either within one `g` field or because the `g` field is also duplicated), then SpaceXpanse will ignore all moves except the last. In other words, in the example above, only `move b` would be "the" move for `g/my game` in this transaction.

```
{
  "g": {"my game": {"x": 1, "x": 2}}
}
```

If keys are duplicated *within a move*, then the data will be passed on like that (i.e. `{"x": 1, "x": 2}`) as move to the game. Games must be prepared for this situation, and handle it gracefully in some well-defined way.

For instance, `libspacexpansegame` will de-dup keys (keeping only the last within each JSON object) before passing data on to the individual game logic.

Admin commands

For an admin command, the `cmd` field can also be duplicated:

```
{
  "cmd": "first",
  "cmd": "second"
}
```

In this case, all associated values are considered valid admin commands for the game in question.

SpaceXpanse ROD transactions in games and other applications

Games may also need to process ROD transactions, at least in a limited fashion. For instance, it may be possible to buy in-game items from the game developer with ROD, or the game may implement a player-to-player market place where transactions are settled in ROD.

To facilitate this, game engines can process all currency outputs created by name transactions that reference their game ID. In other words, a player can issue a move and *in the same SpaceXpanse transaction* also send ROD to, for instance, the SpaceXpanse address of the game developer or a trading partner. The game engine will then (but only if done in the same transaction) know about this, and can implement rules that update the game state accordingly. For instance, a game rule could be like this:

If the move data contains `{"buy": "diamond sword"}` and at least 1,000 ROD are sent to the company address Cxyz, then the player gets a diamond sword for her avatar in the game state.

Note that there exists a second possibility for trading in games: Atomic name updates can be used to couple game moves (e.g. an explicit command to transfer an item) with payments in ROD. This is particularly useful for market places between players themselves, rather than payments to a game developer.

An alternative to handling ROD payments is to require burning of ROD. This can be used to create a cost for certain actions (e.g. to discourage spamming them), while not creating a privileged instance (the receiver of payments). To burn ROD in relation to a given move in a game, they should be sent to an `OP_RETURN` output with `g/GAMEID` as the data part. (The burn must be related uniquely to a specific game, since otherwise a single burn of ROD could be used for multiple games at the same time.)

Processing backwards in time

While the typical behavior of the SpaceXpanse blockchain is to attach blocks on top of each other, it may happen that the blockchain is *reorganized*. This means that one or multiple blocks that were previously part of the best chain must be detached, and alternative blocks are attached to replace them. To handle these situations, game engines need a way to restore old game states or, in other words, process backwards in time when a block is detached.

There are two suggested strategies that may be employed for this, depending on the particular properties of each game:

Archiving old states

The most straight-forward solution is to simply keep "archived copies" of old game states, keyed by the block hashes they correspond to. Then in case of a reorg, the game engine can restore the archived copy of some game state before the chain fork, and update it forward in time with all the blocks that are then attached on top of it.

Note that it is not necessary to store *every* previous game state, as one can simply process forward all following blocks. This means that a good strategy may be to store a game state for the last `n` blocks to cheaply handle short reorgs, and then one game state every `k` blocks for cases where very old game states may be needed (requiring to process, in the worst case, `k - 1` forward steps).

This option is easy to implement because it reuses the forward-processing function `f` that the game engine needs anyway. On the other hand, it requires storing at least a certain number of old game states, which may be costly (or not). Also, if not every game state is archived, this may require more processing power to retrieve an old game state.

Depending on how the current game state is stored, it may be possible to take advantage of existing snapshotting and/or copy-on-write functionality of the underlying file system or database, which can make this particularly easy and cheap to implement.

This model is used successfully in Huntercoin.

Backwards processing and undo data

An alternative is to implement separate logic that reverts the changes made to the game state by a particular block. This can then be used to compute the old game state when detaching a block, before attaching a new one.

Since the moves made in a particular block are archived already by the SpaceXpanse daemon as part of the block, they are readily available when detaching the block. However, it may very well be the case that the forward-processing function `f` is not reversible, because it "destroys" or "overwrites" some data in the old game state. For instance, if a move simply says to "make this boat in my fleet the flagship", then it may not be possible to determine afterwards which boat was the flagship before. To handle these cases, all of the destroyed information must be stored in some extra undo data associated with the block, such that the mapping from the old state and moves to the new state *and the undo data* can be reverted by the backwards-processing function

`b`: (new state, moves, undo data) -> old state.

It is the responsibility of the game engine and not the SpaceXpanse daemon to keep all the undo data relevant for the particular game!

This method is, in some sense, a custom implementation of incremental snapshots for all old game states. It requires more work to implement, but can be the most efficient solution, as the exact nature of the undo data can be optimized for the particular game.

This approach is employed by Bitcoin Core to handle changes to the UTXO set during reorgs.