RPS and Tournament API for the Peerplays Blockchain

This overview document describes the API for Rock-Paper-Scissors (RPS) and Tournaments on the Peerplays Blockchain.

OBJECTS

Information about the rock-paper-scissors game on the blockchain is broken down into three types of objects: *tournaments*, *matches*, and *games*. For rock-paper-scissors, a *game* is the smallest unit, a contest between two players where each player throws a gesture, and the game results in either a win by one of the players or a tie. A *match* is a series of one or more games between two players which results in either a winner or a tie. A *tournament* is a competition between two or more players to find a single winner by playing a series of matches.

These objects are never stored in blocks or transmitted in transactions, but are used for tracking the state of tournaments. They can be accessed by wallets using the `get_objects` RPC call.

Tournament Objects
The information about a tournament stored on the blockchain is divided into two objects. `tournament_object`s have the most-commonly used, less-frequently changing data about the tournament. Tournament objects are assigned ids in the `1.16.x` namespace, and look like this:

~~~.cpp
   struct tournament_object
   {
      /// the account that created this tournament
      account_id_type creator;

      /// the options set when creating the tournament
      tournament_options options;

      /// If the tournament has started, the time it started
      optional<time_point_sec> start_time;

      /// If the tournament has ended, the time it ended
      optional<time_point_sec> end_time;

      /// Total prize pool accumulated
      /// This is the sum of all payers in the details object, and will be
      /// registered_players.size() * buy_in_amount
      share_type prize_pool;

      /// The number of players registered for the tournament
      /// (same as the details object's registered_players.size(),
here to avoid
      /// the GUI having to get the details object)
      uint32_t registered_players;

      /// A reference to an object containing detailed information on
```
~~~

this tournament
      tournament_details_id_type tournament_details_id;

      /// The current state of the tournament
      /// (accepting_registrations, awaiting_start, in_progress,
registration_period_expired, concluded)
      tournament_state state;
  };
~~~
The `options` structure contains the parameters for the tournament chosen by the creator when they created the tournament. It looks like this:
~~~.cpp
   struct tournament_options
   {
      /// If there aren't enough players registered for the tournament
before this time,
      /// the tournament is canceled
      fc::time_point_sec registration_deadline;

      /// Number of players in the tournament.
      uint32_t number_of_players;

      /// Each player must pay this much to join the tournament. This
can be
      /// in any asset supported by the blockchain. If the tournament
is canceled,
      /// the buy-in will be returned.
      asset buy_in;

      /// A list of all accounts allowed to register for this
tournament. If empty,
      /// anyone can register for the tournament
      flat_set<account_id_type> whitelist;

      /// If specified, this is the time the tournament will start
(must not be before the registration
      /// deadline). If this is not specified, the creator must
specify `start_delay` instead.
      optional<fc::time_point_sec> start_time;

      /// If specified, this is the number of seconds after the final
player registers before the
      /// tournament begins. If this is not specified, the creator
must specify an absolute `start_time`
      optional<uint32_t> start_delay;

      /// The delay, in seconds, between the end of the last match in
one round of the tournament
      /// and the start of all the matches in the next round.
      /// NOTE: this is currently unsupported; matches start as soon
as possible
      uint32_t round_delay;

      /// The winner of a round in the tournament is the first to
reach this number of wins.

```
      /// a "best of five" tournament would have number_of_wins = 3
      uint32_t number_of_wins;

      /// Metadata about this tournament.  This can be empty or it can
contain any keys the creator desires.
      /// The GUI will standardize on displaying a few keys, likely:
      ///  "name"
      ///  "description"
      ///  "url"
      fc::variant_object meta;

      /// Parameters that are specific to the type_of_game in this
tournament
      /// The type stored in this static_variant field determines what
type of game is being
      /// played, so each different supported game must have a unique
game_options data type
      game_specific_options game_options;
   };
~~~
And the `game_options` will be a structure like this:
~~~.cpp
   struct rock_paper_scissors_game_options
   {
      /// If true and a user fails to commit their move before the
time_per_commit_move expires,
      /// the blockchain will randomly choose a move for the user
      bool insurance_enabled;
      /// The number of seconds users are given to commit their next
move, counted from the beginning
      /// of the hand (during the game, a hand begins immediately on
the block containing the
      /// second player's reveal or where the time_per_reveal move has
expired).
      /// Note, if these times aren't an even multiple of the block
interval, they will be rounded
      /// up.
      uint32_t time_per_commit_move;

      /// The number of seconds users are given to reveal their move,
counted from the time of the
      /// block containing the second commit or the where the
time_per_commit_move expired
      uint32_t time_per_reveal_move;
   };
~~~
The `tournament_object` has a pointer to a `tournament_details_object`
which keeps the more rapidly-changing information about a tournament
that you would need to be aware of if you were participating in or
monitoring the tournament. `tournament_detals_object`s are assigned ids
in the `1.17.x` namespace, and look like:
~~~.cpp
   struct tournament_details_object
   {
   public:
```

```
      /// the tournament object for which this is the details
      tournament_id_type tournament_id;

      /// List of players registered for this tournament
      flat_set<account_id_type> registered_players;

      /// List of payers who have contributed to the prize pool
      flat_map<account_id_type, share_type> payers;

      /// List of all matches in this tournament.  When the tournament
starts, all matches
      /// are created.  Matches in the first round will have players,
matches in later
      /// rounds will not be populated.
      vector<match_id_type> matches;
   };
~~~
### Match Objects
Information about a match is stored in the `match_object`, in the
`1.18.x` namespace.
~~~
  struct match_object
  {
      /// The tournament this match is a part of
      tournament_id_type tournament_id;

      /// The players in the match
      vector<account_id_type> players;

      /// The list of games in the match
      /// Unlike tournaments where the list of matches is known at the
start,
      /// the list of games will start with one game and grow until we
have played
      /// enough games to declare a winner for the match.
      vector<game_id_type> games;

      /// A list of the winners of each game.  This information is
      /// also stored in the game object, but is duplicated here to
allow displaying
      /// information about a match without having to request all game
objects
      vector<flat_set<account_id_type> > game_winners;

      /// A count of the number of wins for each player
      vector<uint32_t> number_of_wins;

      /// the total number of games that ended up in a tie/draw/stalemate
      uint32_t number_of_ties;

      // The winner of the match.  If the match is not yet complete,
this will be empty
      // If the match is in the "match_complete" state, it will
contain the
      // list of winners.
```

```
      // For Rock-paper-scissors, there will be one winner, unless
there is
      // a stalemate (in that case, there are no winners)
      flat_set<account_id_type> match_winners;

      /// the time the match started
      time_point_sec start_time;

      /// If the match has ended, the time it ended
      optional<time_point_sec> end_time;

      /// The current state of the match,
      /// (waiting_on_previous_matches, match_in_progress, match_complete)
      match_state state;
   };
~~~
### Game Objects
~~~.cpp
   struct game_object
   {
   public:
      /// The id of the match this game is a part of
      match_id_type match_id;

      /// The list of players in this game
      vector<account_id_type> players;

      /// When complete, the list of winners of this game
      flat_set<account_id_type> winners;

      /// details about this specific type of game
      game_specific_details game_details;

      /// For internal use, the next time the blockchain needs to
perform an action; this is used
      /// for storing the time commit or reveal moves must be received by.
      fc::optional<time_point_sec> next_timeout;

      /// The state of the game
      /// (expecting_commit_moves, expecting_reveal_moves, game_complete)

      game_state state;
   };
~~~
The `game_details` is a structure with data necessary to run this
specific type of game (a rock-paper-scissors game)
~~~.cpp
   struct rock_paper_scissors_game_details
   {
      /// The commit moves made by each player (players are in the
same order as they are in game_object.players)
std::vector<fc::optional<rock_paper_scissors_throw_commit> > commit_moves;
      /// The reveal moves made by each player
std::vector<fc::optional<rock_paper_scissors_throw_reveal> > reveal_moves;
   };
```

~~~

OPERATIONS

The rock-paper-scissors game extends the BitShares blockchain by adding three main operations:

`tournament_create_operation`

~~~.cpp
   struct tournament_create_operation : public base_operation
   {
      account_id_type creator; /// The account that created the tournament
      tournament_options options; /// Options for the tournament
   };
~~~

This operation is supported by the `tournament_create` CLI wallet command:

~~~
tournament_create creator tournament_options broadcast
ex: tournament_create alice { "registration_deadline":
"2016-11-30-T00:00:00", "number_of_players": 4, "buy_in": { "amount":
10000, "asset_id": "1.3.0" }, "whitelist": [], "start_delay": 10,
"round_delay": 10, "number_of_wins": 2, "meta": {}, "game_options": [
0,{ "insurance_enabled": false, "time_per_commit_move": 10,
"time_per_reveal_move": 5, "number_of_gestures": 3 } ] } true
~~~

`tournament_join_operation`

This operation is used to register a player for a tournament.

~~~.cpp
   struct tournament_join_operation : public base_operation
   {
      /// The account that is paying the buy-in for the tournament, if
the tournament is
      /// canceled, will be refunded the buy-in.
      account_id_type payer_account_id;

      /// The account that will play in the tournament, will receive
any winnings.
      account_id_type player_account_id;

      /// The tournament `player_account_id` is joining
      tournament_id_type tournament_id;

      /// The buy-in paid by the `payer_account_id`
      asset buy_in;
   };
~~~

This operation is supported by the `tournament_join` CLI wallet command:

~~~
tournament_join payer player tournament_id buy_in_amount buy_in_asset
broadcast
ex: tournament_join alice alice 1.16.0 .1 TEST true
~~~

`game_move_operation`

When it is a player's turn in a game, the player will generate a
~~~

`game_move_operation`:
~~~.cpp
struct game_move_operation : public base_operation
   {
      /// the id of the game
      game_id_type game_id;

      /// The account of the player making this move
      account_id_type player_account_id;

      /// the move itself, depends on the type of game
      game_specific_moves move;
   };
~~~
In rock-paper-scissors, there are two types of move, a commit and a reveal. For a commit, the `move` structure looks like this:
~~~.cpp
   struct rock_paper_scissors_throw_commit
   {
      uint64_t nonce1;
      fc::sha256 throw_hash;
   };
~~~
and for a reveal move, it looks like this:
~~~.cpp
   struct rock_paper_scissors_throw_reveal
   {
      uint64_t nonce2;
      rock_paper_scissors_gesture gesture;
   };
~~~

This operation is supported by the `rps_throw` CLI wallet command:
~~~
rps_throw game_id player_id gesture broadcast
ex: rps_throw 1.19.2 alice rock true
~~~
The `rps_throw` command will commit the player's move, then it will automatically reveal the move at the appropriate time.

`tournament_payout_operation`
There is also a virtual operation that does not appear on the blockchain for payouts. At the end of a tournament, the winner will be paid the winnings, and there will be a new operation in the winner's account history showing that they received their winnings. A rake fee is also paid to the dividend distribution account at the end of a tournament, and a `tournament_payout_operation` will also be generated for that payout.

In the event a tournament is canceled because enough players don't register before the registration deadline, all payers will have their buy-ins refunded, and will get a `tournament_payout_operation` informing them.
~~~.cpp
  struct tournament_payout_operation
   {
~~~

```
      /// The account receiving the payout
      account_id_type payout_account_id;

      /// The tournament generating the payout
      tournament_id_type tournament_id;

      /// The payout amount
      asset payout_amount;

      /// The type of payout
      /// (prize_award, buyin_refund, rake_fee)
      payout_type type;
   };
~~~
```

PARAMETERS

There are several chain parameters the committee can change to control how tournaments work. With the exeption of `rake_fee_percentage`, these chain parameters are used at the time the tournament_create_operation is evaluated to limit what options can be set on newly-created tournaments.

```
~~~.cpp
      uint16_t rake_fee_percentage           ///< part of prize paid
into the dividend account for the core token holders

      uint32_t min_round_delay               ///< minimal delay
between rounds of matches in a tournament
      uint32_t max_round_delay               ///< maximal delay
between rounds of matches in a tournament
      uint32_t min_time_per_commit_move      ///< minimal time to
commit the next move
      uint32_t max_time_per_commit_move      ///< maximal time to
commit the next move
      uint32_t min_time_per_reveal_move      ///< minimal time to
reveal move
      uint32_t max_time_per_reveal_move      ///< maximal time to
reveal move
      uint32_t maximum_registration_deadline ///< value registration
deadline must be before (seconds from time of tournament creation)
~~~
```