

Report

# Project 2 - Colosseum Protocol

---

Rajat Singhal, CS17B042

Kshitij Deogade, CS17B104

Dantala Naga Sai Ram, CS17B010

Anirudh Shivapuja, CS17B002

Prashanth T, CS17B105

## Program Overview

For our project, we have created a semi-implementation of the Colosseum protocol in Python. The program continuously creates tournaments with the same set of 64 nodes till it gets terminated. Relevant messages for the ongoing matches are also printed.

There is a global dictionary of IDs, which are integers here, to corresponding Node object, and is a global variable, as is assumed to be known to all nodes.

Node is checked for **qualification** by checking -

- 1) If it has played round earlier in the tournament
- 2) If it's `current_score = round_no`
- 3) if it has had no keeper yet, if the `round_no` is 0.

There is hash with every player of -

- 1) **Score\_keeper**, which is the mapping of every score of a player in a tournament, to the ID of the keeper it had in that round win.
- 2) **Player\_score**, which is the mapping of [(player\_id,score) : PoW object]. This info is present with the keeper. So, given a keeper, you can query (player\_id,score), to check if it indeed has the PoW.
- 3) **Score\_validator**: Mapping of score of node to Validator node ID

As there are 64 players, and each tournament has  $\alpha=4$  rounds, so players successively in every round are- 64->32->16->8->4 . At the end, the 4 players winning the tournament are selected to propose blocks in Colosseum.

Every node iteratively selects its partner player as a random node, as long as its not itself.

Keepers are allotted for both, and both keepers check if their corresponding nodes are eligible to play match.

- 1) **Byzantine case** - This has been set up such that with 0.2 probability, such that of selected nodes  $i$  and  $j$  to play game, a winner is selected from amongst  $i$  and  $j$  with 0.5 probability, and winner generates a fake PoW, and updates its `score_keeper[]` and `player_score[]` of its keeper.

But immediately after this happens, as its given in Colosseum that a node can be queried anytime, so a random **verifier** node is selected. **Self.\_\_validator\_pow\_list** is a **private** list, inside the validator within which game is played, and it is of the form-

(**tournament\_no,round\_no,winner\_id,loser\_id**). This is added to the list only inside the validator, when the game is played, so in case of unfair play as in above case, this list is never updated inside validator. So, i and j can both give the validator id using **score\_validator hash**, and using this id, validator can be queried if its self.\_\_validator\_pow\_list has got (tournament\_no,round\_no,i,j) or (tournament\_no,round\_no,j,i). If not, then no match was played. Thus, Byzantine case is resolved. **Bad\_score** of winner amongst i and j is incremented here.

- 2) **Non-Byzantine case- Validator** node is randomly selected, given player nodes i and j. Play\_game method is called to play matches between nodes. Here too, verifier checks match result, but it returns true as this was a fair match. Note that **keepers** check for **validity** of nodes, just so that valid players can be found for a match/round, whereas **verifiers** check the **outcome** of the match.


Each tournament involves the selection of a pair of nodes who haven't played a match yet in the current round, with distinct keepers chosen for each player. The keepers perform the necessary validation checks, and if the players pass, a validator is assigned and the match is played. In an authorized match, the validator is responsible for computing the result of the match and forwarding the *PoWin* to the players and their keepers. To also simulate byzantine behaviour, either of the nodes is set to have won the match (with probability 0.2). This is later caught by the validator when it performs its checks, and the nodes that exhibited the behaviour are tracked (used in computing the player score). At the end of a tournament, the IDs of the qualified nodes as well as the number of qualified nodes are printed. In our program we use 4 as our *alpha* value, and so we expect to see 4 qualified nodes.

## Players

Each node is made to keep track of the number of times it has exhibited byzantine behaviour (stored in **bad\_count**). The player's score is computed as:

$$\text{total\_score} / \text{abs}(\mathbf{a} - \mathbf{v}) * \text{bad\_count}$$

where **total\_score** is the sum of the player's scores over all previous tournaments, **a** and **v** are the random values temporarily assigned to the player and the validator respectively.



In this manner, the player is incentivized to win matches as well as penalized for reporting incorrect results. Note that as `total_score` is taken, the total stake of a player across all tournaments has been taken.

## Keepers

A player's keeper checks if the node has played the round earlier, and also checks if the player's **`curr_score`** is consistent with the **`round_no`**. It also checks if the keeper for the current round is assigned correctly. Next, it goes through each of the player's records from its previous rounds and cross checks with the corresponding keeper and *PoWin*. If the player fails any of these checks, then the method **`keeper_checks_if_player_valid()`** returns **`False`** and a new pair of nodes is selected for playing the next match.

## Validators

Along with the keepers, the validators perform their own checks on the players. Validators compute the result of the match by calling the **`play_game()`** method, the result of which is appended to the validator's own list of results: **`__validator_pow_list`**. Thus, the inaccurate results reported by byzantine players can easily be identified as the validator for each round needs to be mentioned as well.

## Proof-of-Win

*PoWin* is internally represented as an object that contains the **`tournament_no`**, **`round_no`**, **`winner`**, **`loser`** of its corresponding match.



## Individual contribution

1. Kshitij (CS17B104):- **stake\_in\_colosseum\_\_byzantine\_case.py** - Wrote full code ; Idea of how to add stake, how to implement the program, how to add Byzantine case;  
Discussion\_of\_ideas
2. Rajat(CS17B042):- **stake\_in\_colosseum\_non\_byzantine\_case.py**- Wrote full code +  
Discussion\_of\_ideas
3. Prashant(CS17B105):- Report Writing + Discussion\_of\_ideas + Understand and comment code
4. Nagasairam (CS17B010) :- Debugging + Testing code + Corner cases +Ideas from Colosseum paper
5. Anirudh Shivpuja (CS17B002):- Debugging+Testing code + Corner cases + Ideas from Colosseum paper

