Challenger: Blockchain-based Massively Multiplayer Online Game Architecture

Boris Chan Yip Hon[1], Bilel Zaghdoudi, Maria Potop-Butucaru, Sébastien Tixeuil, and Serge Fdida

Sorbonne Universite, CNRS, LIP6, F-75005 Paris, France
Name.Surname@lip6.fr

Abstract. We propose Challenger a peer-to-peer blockchain-based middleware architecture for narrative games, and discuss its resilience to cheating attacks. Our architecture orchestrates nine services in a fully decentralized manner where nodes are not aware of the entire composition of the system nor its size. All these components are orchestrated together to obtain (strong) resilience to cheaters. The main contribution of the paper is to provide, for the first time, an architecture for narrative games agnostic of a particular blockchain that brings together several distinct research areas, namely distributed ledgers, peer-to-peer networks, multiplayer-online games and resilience to attacks.

Keywords: multiplayer online games, peer-to-peer architecture, blockchain, cheating resilience

1

1 Introduction

A video game is an electronic game involving human-machine interaction and visual feedback. Multiplayer games feature players, either competing or collaborating in Player Versus Player or Player Versus Environment modes. Massively Multiplayer Online Games (e.g. World of Warcraft) support hundreds to thousands of players online, with or without direct interaction. Traditionally, these games were designed in a client-server mode. However, the scalability of this architecture becomes harder and harder to maintain because of the bottleneck points at the server level. To prevent this, game editors provision resources (e.g. bandwidth, hardware, replication facilities) to quickly absorb spikes in the number of players. To avoid the inherent cost of the provisioning peer-to-peer architectures have been recently explored. Their major advantage is the distribution of computational tasks across nodes, with each node serving as both server and client. Even though this design philosophy seemed to be the holy grail, P2P architectures became quickly the target of various attacks and cheating behaviours [29] and [21] at the game level, application level, or protocol level. For example, some attacks target the

[1] Acknowledgements: The work presented in this document has received funding from the EU Horizon Europe research and innovation Programme under Grant Agreement No. 101070118.

interruption of information dissemination having as a consequence confusing players about the game's current stage. Other attacks perform illegal game action hence allowing clients to unfairly manipulate the game state, bypassing its physical laws. Finally, unauthorized access to information yields to the exploitation of undisclosed information. Academic research struggled for decades to study and propose solutions for the cheating methods reviewed in [29]. Initial publications aim to enhance or devise algorithms for cheating detection, such as dead reckoning [28], area of interest in Emanuele et al. [5], and secure referee selection problem by Webb et al. [22]. Subsequent works [28], [19] broaden the platforms including mobile devices. Additionally, advancements in signature protocols [11] and the introduction of new hybrid architectures (e.g. [28], [17], [15]) aim to address attack vectors through design improvements like trust between peers, information disclosure, and pair clustering. Recently, the resurgence of studies (e.g. [6], [18]) on specific attack vectors has occurred through software utilization and artificial neural networks. Despite the prolific work on preventing cheating in multiplayer online games, cheating still prevails and evolves in online games. The recent advances in distributed ledgers opened a new direction of research interesting to be explored in the context of cheating detection and prevention in multiplayer online games. Blockchain technology is a decentralized product of distributed systems, featuring consensus mechanisms, immutability, security, data persistence, transparency, and scalability.

2 Related Work

Since the publication of the seminal papers in blockchain area [12] (Bitcoin) and [23] (Ethereum) various works have explored blockchain-based architectures for Massively Multiplayer Online Games. For example, in [14] the authors introduced GiNA, a blockchain-based gaming scheme leveraging packet transfer schemes for security and authenticity. The authors of [10] proposed a blockchain-based cheat prevention and robustness mechanism for Massively Multiplayer Online Game. In [20] the authors presented a decentralized authoritative multiplayer architecture, focusing on cheat detection at protocol and architecture levels. The authors of [32] introduced a blockchain consensus model based on the Bryllite Consensus Protocol, supporting a hyper-connected Massively Multiplayer Online Game ecosystem and enabling user participation through Proof of Participation. In [31, 25, 24] the authors proposed Proof of Play, a consensus model for blockchain-based Peer to Peer gaming systems, showcasing a serverless match-based online blockchain game, Infinity Battle. However, none of these works proposed a generic architecture for Massively Multiplayer Online Games agnostic to a specific blockchain and none of them address narrative games.

Blockchain technology is not universally suitable for all Massively Multiplayer Online Game genres; only a select few meet its requirements [1, 2, 26] due to the discrepancy between latency requirements for MMOG and validation time needed for each block publication in blockchains. However, narrative adventure games played in real-time, where actions are spaced by brainstorming phases, align well with the flow of time in blockchain architecture. Their latency and Action Per Minute values closely match current blockchain metrics like transactions per second (TPS) and block confirmation time.

Our contribution. We propose Challenger, a blockchain-based peer-to-peer architecture for narrative games. We detail the API of the necessary services and examine cheating scenarios to test the resilience of services and the impact of attacks on the Challenger architecture. Interestingly, our design is agnostic to particular blockchains. Any blockchain that supports smart contracts can be used as an underlying layer.

3 Challenger: Modular Decentralized Blockchain-Based Architecture for Narrative Games

3.1 Narrative online games

Narrative adventure games resemble interactive movies and visual novels, presenting real-time, pre-scripted scenes that change based on player interaction. These games feature branching narratives, where player choices influence events, creating a personalized story. Gameplay elements, such as conversation trees, puzzles, and Quick Time Events, support story immersion. In a narrative game context, inspired by Detroit: Become Human [7], the player assumes a detective role, investigating a crime scene. The crime scene serves as the level, with evidence found representing progress items. Exploring the crime scene and finding evidence moves the investigation forward, with different story branches based on the evidence found. The DAG (directed acyclic graph) in figure 1 shows a level with various possible scenarios. The player starts at the left end, with starting points at nodes a, b, or c, depending on in-level actions. The level ends at the right, with different realization scenarios represented by leaves i, l, m, q, r, and s.

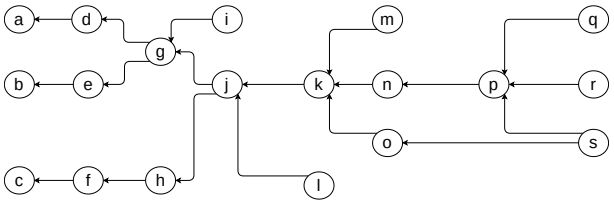

Fig. 1: Chapter scheme for a narrative game

Intermediate nodes between start and end nodes represent interactive game sequences where the player interacts with the game. These interactions may include quest resolution, research periods, action sequences, or multiple-choice discussions. The player's actions and choices alter the level's outcome. For instance, leaf i is an end scenario accessible only if the player started at node a or b. The node chain to access leaf i is either a, d, g, i or b, e, g, i. It is impossible to activate leaf i from starting node c. Choosing a leaf on the DAG selects a possible scenario end for the current level. Players can return to a previous level to unlock actions for other scenarios on subsequent levels. Once an action or scenario is completed, it cannot be revisited unless the Recovery service is called following a path block identified by the Conflict service. This means the

user must restart at a specific previous level to begin the next level as desired, having made the correct choices to achieve the desired starting node (a, b, or c).

The game state saved in the blockchain represents a path from the beginning of the level to a leaf of the DAG (Figure 1) (e.g., a, d, g, i or a, d, g, j, k, n, p, r or c, f, h, j, k, o, s). The distributed ledger monitors the game, detects wrong behaviour, and provides a global log for analyzing behaviour and adapting the game experience. Each verification service call is recorded on the blockchain. The storage service saves local progress within the level, focusing on level-up information from a to b.

3.2 Challenger architecture

This architecture relies on blockchain technology, utilizing its data structure to retrieve the history of operations recorded on the blockchain. In narrative games, this means game states and transitions are written on the blockchain. Unlike traditional client-server architectures, where game progress is stored either on the server or the client—both imposing computational burdens that can impact client-side performance and server scalability—the Challenger architecture uses the blockchain for storage and employs a peer-to-peer network for more scalable exchanges. This decentralized approach enhances both scalability and surveillance. Decentralization eliminates single points of failure, enhancing resilience and security. Peer-to-peer networks can handle larger volumes of data without centralized bottlenecks, and blockchain's immutable ledger provides a transparent record of operations. However, designing and maintaining decentralized systems can be more complex than traditional architectures, and blockchain operations can be slower and more resource-intensive, leading to higher costs. Non-blockchain alternatives include Trusted Execution Environments (TEEs), which are secure areas in processors that run code in an isolated environment, offering high security and reducing the need for extensive cryptographic proofs. However, they have limited scalability and depend on hardware. Distributed consensus protocols, which are algorithms ensuring agreement among distributed systems, offer enhanced fault tolerance and flexibility in design but can result in slower consensus times and complex implementation. Cryptographic methods, such as zero-knowledge proofs and secure multi-party computation, provide strong privacy guarantees and reduced trust assumptions, but they come with high computational overhead and complexity in integration.

The Challenger architecture (Figure 2) consists of nine essential services. Each peer provides all essential services, but some services are local, while others are distant. Local services include Membership (Algorithm 1), Publication (Algorithm 3), Conflict (Algorithm 4), Storage (Algorithm 7), Anomaly (Algorithm 9), and Orchestration. Distant services are Verification (Algorithm 2), Recovery (Algorithm 6), Choice (Algorithm 5), and Misbehaviour (Algorithm 8).

Challenger presents seven different scenarios, which describe the basic exchange between Orchestration and the architecture. The different use cases and their respective within the services presented in Figure 2 are :

- Process for registering : 1a, 2a, 3a, 4a
- Process for altering the game state
  - Conflict (Algorithm 4): 5b, 6b, 7b, 8b, 9b, 10b, 11b

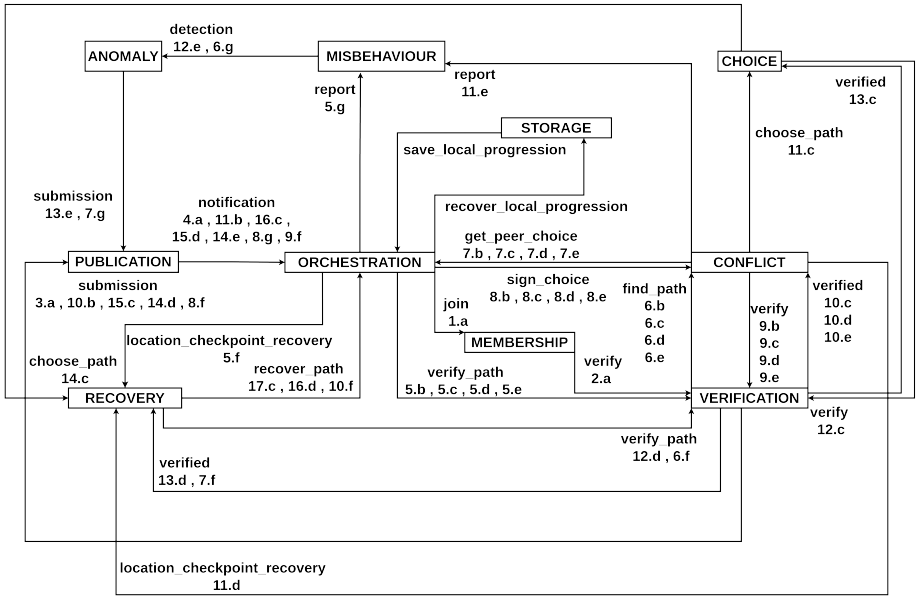

Fig. 2: Architecture Major component

- Choice (Algorithm 5) : 5c, 6c, 7c, 8c, 9c, 10c, 11c, 12c, 13c, 14c, 15c, 16c, 17c
- Recovery (Algorithm 6)
  - request by Conflict service (5d, 6d, 7d, 8d, 9d, 10d, 11d, 12d, 13d, 14d, 15d, 16d)
  - request by the user (Algorithm 6) : 5f, 6f, 7f, 8f, 9f, 10f

– Process for reporting misconduct (Algorithm 8)
- request by Conflict service (5e, 6e, 7e, 8e, 9e, 10e, 11e, 12e, 13e, 14e, 15e)
- request by a user (5g, 6g, 7g, 8g)

4 Challenger services in detail

We discuss the implementation of the services, Figure 2. For each service, we provide a pseudocode representation of the service's implementation, an overview of its architecture, and a sequence diagram demonstrating a specific use case. This approach helps to clearly illustrate the functionality and operation of each service.

4.1 Process for registering

Membership service Membership service facilitates the initial user connection to the game by assigning a unique user identifier (id_user) and game identifier (id_game) in exchange for the client identifier. These IDs are necessary for game participation and are generated using the client identifier as a seed. The user follows steps 1a, 2a, 3a, and

Algorithm 1 Membership Service

```
1: Local variable :
2: IDENTIFICATION a table of (id_user, id_game) clients register
3: Function :
4: - generate (integer): based on integer generates a random integer
5: - add (integer 1, integer 2): add a row of arguments in the IDENTIFICATION table
6: - remove (integer 1, integer 2): remove the row including arguments of IDENTIFICATION table
7:
8: Upon receiving Join(id_client) from Orchestration do
9:     if id_client not in IDENTIFICATION then :
10:        id_user = Hash(generate(id_client))   ▷ generate id_user based on id_client, this allow a mapping between
   id_client and id_user, in order to be able to identify user's id_client
11:        add(id_user, id_game)   ▷ add id_user and id_game to the table of register
12:        deliver (id_user, id_game) to Orchestration
13:        deliver (id_user, id_game) to Publication service
14:    else :
15:        deliver (id_user, id_game) to Orchestration
16:    endif
17: enddo
18: Upon receiving Unsubscribe(id_user, id_game) from Orchestration do :
19:    if (id_user, id_game) in IDENTIFICATION then :
20:        remove (id_user, id_game)
21:    else :
22:        deliver NO_EXISTENT_USER_ID to Orchestration
23:    endif
24: enddo
```

4a (Figure 2) to utilize the Membership service and obtain the IDs. Once the IDs are generated, Orchestration service informs the user of its assigned IDs. These IDs can either be stored locally or queried each time the user connects to the game. For this service, we assume that each tuple of (id_user, id_game) is stored locally in the user application.

- membership.join(id_client): join is called, and the client's ID (id_client) is used as a seed to generate the user ID (id_user).
- publication.submission((id_user, id_user, current_level, next_level): submission is called to submit the newly generated player IDs to the ledger.
- orchestration.notification(new_publication_on_ledger): the user receives a notification to update the local storage of users with the new IDs.
- orchestration.update() : the user updates his local storage

Once the IDs have been successfully verified and sent to the Publication service, the user's Orchestration receives a notification from both services. This allows other users to update their available user table and informs them of the new client's IDs, ensuring that all users are up-to-date with the latest information.

Verification service Verification service ensures the validity of a user's request to progress to the next level by evaluating if the transition from the current level meets specific conditions. It utilizes local storage to retain game state headers and minimal-level information for this purpose. The primary objective of the Verification service is to filter legitimate requests from illegitimate ones. If the progression from one game state to another is recorded as a local variable and follows the logical path of the game's different states, progression is considered legitimate. Conversely, if the progression deviates from the logical path, it may be indicative of misbehaviour or a cheating attack.

The verification service is designed to detect and prevent such instances, ensuring that all level advancements are valid and adhere to the game's established rules.

We analyze the verification process and its various scenarios, assuming that the Verification call made by Orchestration will be successful and that all components are functioning correctly. We examine different conflict cases, such as when a user faces multiple choices when a user delegates a choice to Choice service, and when a user is blocked and must recover an anterior stage of the game to proceed.

The verification process involves three services, as depicted in Algorithm 2:

- verification.verify_path((id_user, id_user, current_level, next_level): verify_path is called to determine the legitimacy of the level-up by examining the different game states. The call to verification.verify_path can be made by :
  - conflict.find_path(id_user, id_game, current_level, targeted_level)
  - choice.choose_path (id_game, current_level)
  - recovery.recovery_checkpoint (recovery_checkpoint_location)
- publication.submission((id_user, id_user, current_level, next_level): submission is called to write the change of level from the current one to the next as proof of level-up.
- orchestration.notification(new_publication_on_ledger) : notification is called to update the new local level of the user.
- orchestration.update() : the user updates his local storage

The verification service reads the blockchain to assess the legitimacy of the level-up between the different game states. A comparison is made between the requested game state and the one found within the blockchain. If not, a local comparison is performed between the game states stored locally within the Verification service and the requested level. The verification process algorithm, as illustrated in Algorithm 2, has a Verification service that first evaluates the legitimacy of the current level of the user requesting a level-up. Based on the result, users can either advance to the next level or call the Conflict service to find a path from their current level. This scenario indicates that the user's current position is legitimate, but the next level is unreachable due to insufficient game progress.

Publication service Publication service holds the exclusive right to write on the blockchain and is thereby the sole service capable of updating the game's stage within the ledger. This service records the outcome of user requests, which could be derived from various services such as verification, conflict, choice, misbehaviour, or anomaly. It is important to note that the Publication Service is responsible only for publishing the outcome; all verification procedures to confirm the information's accuracy must be completed beforehand. Upon publication by the Publication service and the subsequent ledger update, users can access the blockchain to refresh their local cache. Algorithm 3 illustrates the Publication service algorithm, which outlines several scenarios where the submission is invoked to publish.

4.2 Process for altering the game state

Conflict service Conflict service is employed to resolve two primary issues: scenario ambiguity and a blocked scenario.

Algorithm 2 Verification service

```
Local variable :
ALL_GAME_STATE table of all game states inside the game, it is used to verify the validity of the stage of the game when a client initiates a request to level up to the next level.
Function :
- check(game_state) :
verify if all actions done until the current state of the game is legit
- advance(game_state_1, game_state_2) :
verify if the level up from game_state_1 to game_state_2 is legit.

Upon receiving Verify_path(id_user, id_game, current_level, next_level) from the Conflict, Choice, Recovery, application do :
    if check(current_level) == SUCCESS and advance(current_level, next_level) == SUCCESS then :
        deliver submission(current_level, next_level) to Publication service
    elif check(current_level) == SUCCESS and advance(current_level, next_level) == FAILURE then :
        deliver find_path(id_user, id_game, current_level) to Conflict service
    elif check(current_level) == FAILURE and advance(current_level, next_level) == SUCCESS then :
        deliver Report(id_user, id_game, current_level, next_level) to Misbehaviour service
    else :
        deliver Report(id_user, id_game, current_level, next_level) to Misbehaviour service
    endif
enddo

check(current_level): The idea is to use a recursive function to verify from the last block to the block where id_user and id_game of the client appear for the first time. The condition to stop the function and to unstuck it is to find the block where the client initialized its game.
check(current_level) will call check(precedent_level_before (current_level)), and this one will call check(precedent_level_before (precedent_level_before (current_level))) and so on, until check calls id_user and id_game.
The process of stacking is the verification itself. If the stack goes without any issues, the current stage of the game is legit.
advance(current_level, next_level): Verify if the request to level up from current_level to next_level is within the ALL_GAME_STATE table. If this is the case, the request is legit; otherwise, there is a call to Conflict service.
```

- Scenario ambiguity can be addressed in two ways:
  - The first approach involves presenting the choice to the user and requesting his decision.
  - The second approach entails deciding an algorithm that selects a path for the user.
- In the case of a blocked scenario, the only solution is to recover a previous game stage, specifically one where the scenario is ambiguous. This intersection of scenarios allows the player to alter the outcome of the blocked scenario.

A scenario intersection can be understood as a game state where multiple scenarios are possible, and a change in gameplay can occur if all requirements for a specific scenario are fulfilled. The conflict resulting from multiple choices is resolved by the user's decision, following sequence 5b, 6b, 7b, 8b, 9b, 10b, 11b in Figure 2. This scenario illustrates a case study where a user encounters a choice of path and selects one when prompted by the Conflict service. The user's choice is authenticated using their private key and submitted to Publication.

During the Conflict process (Algorithm 4), all potential paths the user can take are calculated. This step serves as a preventive measure for Recovery service or Choice service, as (a) The user may be on a path leading to a dead-end, in which case Recovery is immediately initiated from a level offering more choices. (b) The user opts to use Choice, and these paths represent the possible directions the user can take within the

Algorithm 3 Publication Service

```
1: Function :
2: - write (object) :
3: writes in the distributed ledger object
4:
5: Upon receiving submission(id_user, id_game, current_level, next_level) from Verification service do :
6:       write(id_user, id_game, current_level, next_level) ▷ write the progression of id_user in id_game from current_level to
   next_level
7: deliver Notification (New_publication_on_ledger) to Orchestration service
8: enddo
9: Upon receiving submission(id_user, id_game, current_level, sign_client_choice) from Conflict service do :
10:      write(id_user, id_game, current_level, sign_client_choice)
11: deliver Notification (New_publication_on_ledger) to Orchestration service
12: enddo
13: Upon receiving submission(id_user, id_game, current_level, Recovered_path) from Conflict service do :
14:      write(id_user, id_game, current_level, Recovered_path)
15: deliver Notification (New_publication_on_ledger) to Orchestration service
16: enddo
17: Upon receiving submission(id_user, id_game, current_level, Chosen_path) from Choice service do :
18:      write(id_user, id_game, current_level, Chosen_path)
19: deliver Notification (New_publication_on_ledger) to Orchestration service
20: enddo
21: Upon receiving submission(Log_anomaly) from Anomaly service do :
22:      write(Log_anomaly)
23: enddo
24:
25: write(id_user, id_game, current_level, Result_path): write what is given in argument in the blockchain. The argument
   of function write() can be the level up from current_level to next_level, the decision sign by the client, a recovered path
   proposed by Conflict service, or a path randomly drawn by the blockchain.
26: write(Log_anomaly): update player's IDs that are designed as cheaters.
```

current level. These paths are forwarded to the Choice service to randomly select one based on the user's current level and paths already in the blockchain through user submissions.

Choice service Choice service is invoked when a user decides to allow the blockchain to determine the path they will take. The set of paths from which one will be selected for the user, known as SET_OF_PATH, depends on the paths already present in the blockchain for the current level. Within the Choice service, the blockchain is read, and all paths previously taken by other users to level up from the current level are gathered in a set called BLOCKCHAIN_PATH. A new set of paths, INTERESTING_PATH, is created by subtracting BLOCKCHAIN_PATH from SET_OF_PATH. Choice service then selects a path for the user from this INTERESTING_PATH set. Choice process follows the sequence 5c, 6c, 7c, 8c, 9c, 10c, 11c, 12c, 13c, 14c, 15c, 16c, 17c Figure 2 and involves the following services: Verification requests level up post and initiates Conflict service; Conflict detects multiple paths and seeks a decision from the user; Orchestration signs the user's decision using their private key; Choice chooses a random path from the blockchain; Publication submits level up with the proposed Choice path; Orchestration updates the user's local level.

Recovery service The recovery service is responsible for restoring a previous state of the game. It receives instructions on which game stage to recover to proceed with the recuperation. There are two ways to invoke the Recovery service:

Algorithm 4 Conflict Service

```
1: Service Conflict is called when a problem is detected, the purpose of this service is to determine the type of the problem
   and send it to the right service for a specific treatment.
2: Local variable :
3: ALL_GAME_STATE_STEP_BY_STEP table of all the game states step by step and arranged by hierarchically level.
4: LOG_ORCHESTRATION table of all requests to orchestration service
5: Function :
6: - following (current_level) :
7: returns a set of all possible following paths starting from current_level
8: - find (current_level) :
9: find a past branching from current_level, where an over scenario can be made
10:
11: Upon receiving find_path (id_user, id_game, current_level, next_level) from Verification service do :
12:     set_of_path = following (current_level) ▷ return a set of all the possible following paths starting from current_level
13:     if set_of_path is empty then : ▷ there is no path starting from the current_level, call of Recovery service to recover
   at a precedent level
14:         find (current_level)                    ▷ find the branching from where an over scenario can be made
15:         deliver location_recovery_checkpoint to Recovery service
16:     else :                                                                    ▷ there is a at least one path
17:         if set_of_path == 1 then:          ▷ there is only one and unique path which derives from the current_level
18:             if unique_path == next_level then:     ▷ the path the client wishes to follow is the same as the one he is
   allowed to take
19:                 deliver submission (id_user, id_game, current_level, unique_path) to Publication service
20:             else:          ▷ this means that the unique path the user has is different from the one he is currently on
21:                 if submission(id_user, id_game, current_level, unique_path) in LOG_ORCHESTRATION :
22:                     deliver submission(id_user, id_game, current_level, unique_path) to Misbehaviour service
23:                 else
24:                     deliver submission(id_user, id_game, current_level, unique_path) to Orchestration service
25:             endif
26:         if set_of_path > 1 then:           ▷ this mean that the client can choose between at least 2 different paths,
27:             ▷ the first solution is to ask the point of view of the user and the second solution is to call Choice service
   which will choose randomly a path for the client
28:             deliver get_peer_choice (id_user, id_game, current_level, set_of_path or random_choice) to Orchestration ▷
   the user is free to make the choice he wishes to follow
29:                 ▷ the user has only access to the set_of_path paths without knowing the rest of the possible scenario
30:         endif
31:     endif
32: enddo
33:
34: Upon receiving get_peer_choice (unknow_decision) from Orchestration do :
35:     if unknow_decision == CLIENT_CHOICE then :
36:         deliver submission (id_user, id_game, current_level, CLIENT_CHOICE) to Publication service
37:     elif unknow_decision == RANDOM_CHOICE then :
38:         deliver Choose_path (id_user, id_game, current_level, RANDOM_CHOICE) to Choice service
39:     elif unknow_decision == CONTINUE then:     ▷ client continues on the same scenario until he will be blocked to
   call Recovery service
40:         deliver submission (id_user, id_game, current_level, CONTINUE : next_level) to Publication service
41:     elif unknow_decision == RECOVERY then: ▷ client calls Recovery service to find a branch from where he will be
   able to change the current_level he currently is on
42:         deliver location_recovery_checkpoint to Recovery service
43:     endif
44: enddo
45:
46: following (current_level): The objective is to locate all the leaf nodes in a Directed Acyclic Graph (DAG). Starting from
   an arbitrary node in the DAG, the function aims to identify all the leaves based on the current position of the node in the
   scenario. A depth-first search algorithm can be employed to find all the leaf nodes.
47: finds (current_level): The function finds searches for an intersection of the scenario that is anterior to the current node. In
   other words, it identifies a parent node of the current node with more than one child node. These child nodes represent the
   intersection within the scenario that determines the ending storyline of the client. To find all specific nodes, a breadth-first
   search algorithm can be used.
```

– Direct call by users: In case they have forgotten their last level and want to recover the entire level without considering any progress.

Algorithm 5 Choice Service

```
Local variable :
BLOCKCHAIN_PATH all paths registered so far given the current level within the blockchain
Function :
- which_path (current_level) :
returns all possible paths to leaves given the location within the current level
- random_choice_path (current_level) :
returns randomly a path not taken by most of the users
Upon receiving Choose_path (id_user, id_game, current_level, next_level) from Conflict service do :
    load BLOCKCHAIN_PATH
     SET_OF_PATH = which_path (current_level)
     random_choice_path (current_level)

deliver submission (id_user, id_game, current_level, random_blockchain_drawn) to Publication service
enddo

which_path (current_level): is the same function as following in Conflict service, see algorithm 4
random_choice_path (current_level) :
SET_OF_PATH are all paths give recoveryrrent_level
BLOCKCHAIN_PATH are paths explored within the blockchain for a given current_level
INTERESTING_PATH = SET_OF_PATH minus BLOCKCHAIN_PATH
return random choice within INTERESTING_PATH
```

– Call by Conflict service: When no path is found during the Conflict service's path-finding process. In this case, Conflict reads the blockchain to find a path ambiguity and saves its location as a checkpoint for Recovery to execute.

We focus on the second scenario involving Conflict, Recovery process follows sequence 5d, 6d, 7d, 8d, 9d, 10d, 11d, 12d, 13d, 14d, 15d, 16d of Figure 2 and involves the following services: Verification requests level-up verification and initiates Conflict service; Conflict: No potential path is found, so a checkpoint is determined to recover a previous game state. The conflict then calls Recovery to recover the checkpoint location and submits the level recovery location to Publication for Orchestration; Publication submits the change of level from the current one to the recovered one; Recovery locates the level to recover and sends it to Orchestration; Orchestration updates both the recovered level and its location for potential future Recovery calls due to the same player judgment error.

In the Recovery Algorithm 6 process, a log of recovery requests made by users and Conflict service is maintained. This log, known as CHECKPOINT LOG, is used to monitor users' level-up progress during their game. This service helps to create a challenging game-play experience based on the requests made by Conflict to unlock users' progression. The location of the recovered game state (recovered_level) can be stored within the Storage service, but for our purposes, we assume that the game state can be stored locally by users.

Storage sercive Storage service, Algorithm 7, enables the saving and retrieval of the current level of a user and its progression without requiring Recovery service every time. At present, Storage service is optional, as we assume that a stage of the game can be loaded and saved within the local cache of the Orchestration service. However, if the game stage can no longer be saved locally within Orchestration, Storage service will become a critical component of this architecture.

Algorithm 6 Recovery Service

```
Local variable :
CHECKPOINT_LOG table of all recovered locations demanded by id_user on id_game and by Conflict service to.
Function :
- add (id_user, id_game, current_level):
add the request in CHECKPOINT_LOG
- extract (level_to_recover_location) :
recover level_to_recover thanks to level_to_recover_location from the distributed ledger

Upon receiving Choose_path (id_user, id_game, current_level, next_level) from Conflict service do :
    add (id_user, id_game, recovery_location)
    extract (id_user, id_game, current_level, recovery_location)
enddo

add (id_user, id_game, current_level):
Recovery service uses a local table to store the player's IDs who request to recover a state of the game. The information
  stored are Players IDs and the stage of the game to recover

extract (id_user, id_game, current_level, requested_level) :
recover the requested level of id_user on id_game. Read the blockchain at current_level and search for requested_level
  at position recovery_location
```

Algorithm 7 Storage Service

```
Local variable :
LOCAL local storage of the client
Function :
- save (id_user, id_game, current_level):
save the current progression
- get (id_user, id_game) :
get the last game state progression saved

Upon receiving Retrieve_local_progression (id_user, id_game) from Orchestration service do :
    get (id_user, id_game)
    deliver Recovered_local_progression
enddo

Upon receiving Save_local_progression (id_user, id_game, current_level) from Orchestration service do :
    save (id_user, id_game, current_level)
    deliver Notification (state_game_saved) to Orchestration service
enddo

save (id_user, id_game, current_level) : locally save the progress of id_user on id_game at current_level game state
get (id_user, id_game) : retrieve last game state saved of id_user on id_game
```

4.3 Process for reporting misconduct

Misbehaviour service The Misbehaviour service is responsible for compiling all behaviour requests made by users or the Conflict service. Its main purpose is to collect a log of requests without any filtering. Misbehaviours can range from a stubborn user repeatedly making the same mistake to trigger a Recovery service, to a user reporting another for tampering with their client code to alter a service's functionality.
The detection process is primarily carried out by Conflict service, which prioritizes misbehaviour collections to ensure none are overlooked. The selection of bad behaviour is done in two steps: (1) Collect misbehaviour reports from users and detection reports from Conflict; (2) Filter by the Anomaly service to differentiate between cheating behaviour and false positives. Conflict service detects suspicious behaviour, such as re-

Algorithm 8 Misbehaviour Service

```
Local variable :
LOG_behaviour table of all misbehaviour detected by Conflict service or reported by users
function :
- add (id_user, id_game, object_of_misbehaving, current_level, next_level):
register report

Upon Receiving Report (id_user, id_game, object_of_misbehaving, current_level, next_level) from Conflict service do :
    add (id_user, id_game, object_of_misbehaving, current_level, next_level)
    deliver Detection (LOG_behaviour) to Anomaly service
enddo

Upon Receiving Report (id_user, id_game, id_user_reported, object_of_misbehaving) from Orchestration service do :
    add (id_user, id_game, id_user_reported, object_of_misbehaving)
    deliver Detection (LOG_behaviour) to Anomaly service
enddo

add (id_user, id_game, id_user_reported, object_of_misbehaving) :
and
add (id_user, id_game, object_of_misbehaving, current_level, next_level) :
compiles the table of misbehaviour, the process focuses on collecting the report, and filtering processing will be done
by Anomaly service.
```

Algorithm 9 Anomaly Service

```
Local variable :
LOG_ANOMALY table of all anomalies detected
Function :
- detection (log_behaviour):
filter log_behaviour to determine cheating behaviour

Upon receiving Detection (log_behaviour) from Misbehaviour service do :
    detection (log_behaviour)
    update LOG_ANOMALY
    deliver submission (LOG_ANOMALY) to Publication service
enddo

detection (log_behaviour) :
analysis of the misconduct to determine real cheating behaviour from stubborn or unwise game-play. LOG_ANOMALY
works as a reference for cheating attacks, the log helps to identify behaviour that seems to be a cheat. LOG_ANOMALY
must be constantly updated to predict potential cheating scheme
```

dundancy, back and forth, or cycles within submission requests.

We describe misbehaviour detection by Conflict service. The collection process of the log follows sequence e, 6e, 7e, 8e, 9e, 10e, 11e, 12e, 13e, 14e, 15e of Figure 2 and involves the following services: Verification request path verification; Conflict: Detects of misbehaviour process; Misbehaviour: Collection of behaviour log; Anomaly filters cheating behaviour from other types; Publication submits log of anomalies; Application updates the local cache with a log of anomalies to deter users from calling the services of users identified as cheaters.
The Misbehaviour Algorithm 8 collects reports made by users or Conflict service to target misbehaviour, referred to as the object of misbehaviour or the reported user ID in cases where a user succeeds in tampering with the client code. The Misbehaviour algorithm serves to collect and deliver the log of misbehaviour for further processing by the Anomaly service.

Anomaly service Anomaly service acts as a filter to differentiate between genuine cheating attacks and false positives. It utilizes a reference dataset, LOG ANOMALY, to identify and classify behaviour, which represents cheating behaviour exhibited by users. After filtering, LOG ANOMALY is submitted to Publication, and an updated log of cheating users is maintained to prevent others from experiencing suspicious outcomes when calling their services.

In the case where a user directly reports misbehaviour to another user, there is a trust issue that must be addressed. The Anomaly process, as depicted in Figure 2, involves the following services: Misbehaviour reports misbehaviour; Anomaly filters cheating attacks from false positives; Publication submits the anomaly log; Application updates and avoids cheating users.

Anomaly filter detection Algorithm 9, takes a log of misbehaviour collected by trusted Conflict services and legitimate reports from altruistic users as input. To filter the misbehaviour log, Anomaly uses a local table of cheating behaviour and reads the blockchain to detect suspicious activity. However, the table of the cheating log used as a reference is biased towards identifying specific patterns. An alternative approach to the filtering problem is to consider the distance between altruistic and rational users from cheating users, shifting the focus from patterns to behavioural distance.

4.4 Challenger Resilience to Cheating

The attacks originate from a combination of studies by Yahyavi et al. and Webb et al. on the classification of cheating attacks. Yahyavi et al. categorize attacks based on their nature, such as interruption of information dissemination, illegal game actions, and unauthorized access to information. Webb et al. classify attacks based on the level at which they occur: game level, application level, protocol level, and infrastructure level. Initially, we draw inspiration from these classifications to determine Challenger's resilience, then examine cheating vectors specific to the Challenger architecture. Table 1 summarizes the cheats for which Challenger is resilient. Challenger is designed to be resilient against a range of cheating attacks by leveraging its blockchain-based architecture and decentralized approach. Cheating attacks can include interruption of information dissemination, illegal game actions, unauthorized access to information, and attacks occurring at various levels (game, application, protocol, infrastructure). The Challenger architecture uses blockchain to maintain an immutable record of game states and transitions, ensuring that all actions are transparent and verifiable by any user in the network. This transparency and decentralization make it difficult for malicious users to alter game states without detection. The system assumes ideal communication conditions, where peer-to-peer communication links are reliable and not compromised by malicious users. However, it does consider the potential malicious behavior of players, including attempts to corrupt services or exploit the blockchain. Challenger's resilience is further enhanced by service separation, requiring user requests to be validated by another user, and decentralized surveillance, which allows any user to verify another user's actions through the game state history provided by the blockchain. In summary, Challenger's architecture provides robust resilience against a wide range of cheating attacks by using a combination of blockchain technology and decentralized peer-to-

Cheat description	Cheating vector	Type of cheat	Altruism required to counter cheating
Impersonation : steal credentials of valuable account	Generate ID client to match id_user and id_game or steal id_user and id_game	(in game layer ; unauthorized information access)	none required : secure by design, the pair (id_user,id_game) is unique for each id_client. A common issue would be to have already someone using the id_user for the same game with a different value of id_game
Theft of a personal item : steal item of id_user on id_game	Extract for the ledger information on personal items	(ledger layer, unauthorized information access)	none required: secure by design, every game state is sign with id_user and id_game of the user's id_user who is playing id_game
Replay cheat : replay a previous state of the game	target verification.verify_path to break Verification during the comparison between the current state and the previous one of the user	(protocol layer, interruption information dissemination)	none required : secure by design, every change in state of game is writing with in the ledger, verification service detect any discontinuity in change of game state
Consistency cheat : a Cheating user act between Byzantine and Altruist to blur the lines of his behaviour	target misbehaviour.report and verification.verify_path in order to publish invalid information on the ledger	(protocol layer, interruption information dissemination)	at least 1 altruist is needed to filter with anomaly.detection() if one user is a real Cheater or not
Suppress correct cheat : suppress a correct state of game once written on the ledger	target publication.submission to break Publication after writing on the ledger	(protocol layer, interruption information dissemination)	none required : secure by design, once a game state, verified or not, is written on the ledger the information is immutable. The case where a game state is not verified will be study in the following attack : corrupt the ledger
Undo : repudiation of a game state	target publication.submission to damage the verification's algorithm	(protocol layer, interruption information dissemination)	None required : secure by design, a state of game can't be cancel, once written by publication.submission, it's stay immutable on the ledger
time cheating : lying at the time a game state is made	target verification.verify_path to ruin Verification during the comparison between the current state and the tampered time level one	(protocol layer, interruption information dissemination	at least 1 Altruist is need to deny verification_path call made by Cheaters
Escaping : cut internet link or shut down IT equipment	target recovery.recovery_checkpoint() to damage the process of retrieving the previous state of game	(infrastructure layer, interruption information dissemination)	non required : secure by design, recovery.recovery_checkpoint() read the ledger to find the latest valid game state writing by publication.submission(). Every state is writing from membership request, to change game state, to misbehaviour, to recovery
Information exposure : information which are not supposed to be disclosed are available due to the transparency on the ledger	exploit the transparency of the ledger to extract useful information	(ledger layer, unauthorized information access)	none required : secure by design, information are available because of the transparency of the ledger. However, only the state of the game and by whom can be read from the ledger, the requirement to unlock such state of game are not disclosed.

Table 1: Vector of cheats ineffective against Challenger

peer verification. This approach ensures that game states are secure, transparent, and verifiable, mitigating the risk of malicious behavior affecting the integrity of the game.

5 Conclusion

In this paper, we propose a modular blockchain-based architecture for peer-to-peer narrative games implementing services using smart contracts. The modular architecture addresses specific functions such as membership, publication, anomaly detection, and game state recovery. We focus on narrative games due to their compatibility with the blockchain's current performance in transactions per second and validation blocks. Our architecture is resilient to a broad range of cheating behaviours.

References

[1] Seyed Mojtaba Hosseini Bamakan, Amirhossein Motavali, and Alireza Babaei Bondarti. "A survey of blockchain consensus algorithms performance evaluation criteria". In: *Expert Systems with Applications* 154 (2020), p. 113385.

[2] Rafael Belchior et al. "A survey on blockchain interoperability: Past, present, and future trends". In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–41.

[3] Glen Berseth and Ravjot Singh. "Asynchronous Real-time Multiplayer Game With Distributed State". In: (2015).

[4] *binance whitepaper*. URL: `https://whitepaper.io/document/10/binance-whitepaper` (visited on 08/22/2023).

[5] Emanuele Carlini, Laura Ricci, and Massimo Coppola. "Integrating centralized and P2P architectures to support interest management in distributed virtual environments". In: *Istituto di Scienza e Tecnologie dell'Informazione (ISTI), CNR, Pisa, Italy, Tech. Rep* (2012).

[6] Subhrajit Chanda, Shaun Star, et al. "Contouring E-doping: A menace to sportsmanship in E-sports". In: *Turkish Online Journal of Qualitative Inquiry* 12.8 (2021), pp. 966–981.

[7] *detroit becom human*. URL: `https://www.quanticdream.com/fr/detroit-become-human` (visited on 08/22/2023).

[8] Stefano Ferretti, Marco Roccetti, and Roberta Zioni. "A statistical approach to cheating countermeasure in P2P MOGs". In: *2009 6th IEEE Consumer Communications and Networking Conference*. IEEE. 2009, pp. 1–5.

[9] Mohsen Ghaffari et al. "A dynamic networking substrate for distributed MMOGs". In: *IEEE Transactions on Emerging Topics in Computing* 3.2 (2014), pp. 289–302.

[10] Sukrit Kalra, Rishabh Sanghi, and Mohan Dhawan. "Blockchain-based real-time cheat prevention and robustness for multi-player online games". In: *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies*. 2018, pp. 178–190.

[11] Dapeng Li, Liang Hu, and Jianfeng Chu. "A more efficient secure event signature protocol for massively multiplayer online games based on P2P". In: *2016 International Forum on Mechanical, Control and Automation (IFMCA 2016)*. Atlantis Press. 2017, pp. 291–299.

[12] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (May 2009). URL: `http://www.bitcoin.org/bitcoin.pdf`.

[13] Jehwan Oh et al. "Bot detection based on social interactions in MMORPGs". In: *2013 International Conference on Social Computing*. IEEE. 2013, pp. 536–543.

[14] Nirav Patel et al. "GiNA: A Blockchain-based Gaming scheme towards Ethereum 2.0". In: *ICC 2021-IEEE International Conference on Communications*. IEEE. 2021, pp. 1–6.

[15] Jared N Plumb, Sneha Kumar Kasera, and Ryan Stutsman. "Hybrid network clusters using common gameplay for massively multiplayer online games". In: *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 2018, pp. 1–10.

[16] *polygon whitepaper*. URL: https://polygon.technology/papers/pol-whitepaper (visited on 08/22/2023).

[17] James Prather, Robert Nix, and Ryan Jessup. "Trust management for cheating detection in distributed massively multiplayer online games". In: *2017 15th Annual Workshop on Network and Systems Support for Games (NetGames)*. IEEE. 2017, pp. 1–3.

[18] Kalpit Sharma and Arunabha Mukhopadhyay. "ANN model for cyber-risk management for DDoS attacks in massively multiplayer online gaming". In: *In bright internet global summit* (2020), p. 2020.

[19] Yuan Tian et al. "Swords and shields: a study of mobile game hacks and existing defenses". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 2016, pp. 386–397.

[20] Aleksandar Tošić and Jernej Vičič. "A Decentralized Authoritative Multiplayer Architecture for Games on the Edge". In: *Computing and Informatics* 40.3 (2021), pp. 522–542.

[21] Steven Daniel Webb and Sieteng Soh. "Cheating in networked computer games: a review". In: *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*. 2007, pp. 105–112.

[22] Steven Daniel Webb, Sieteng Soh, and Jerry L Trahan. "Secure referee selection for fair and responsive peer-to-peer gaming". In: *Simulation* 85.9 (2009), pp. 608–618.

[23] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[24] Feijie Wu et al. "Facilitating Serverless Match-based Online Games with Novel Blockchain Technologies". In: *ACM Transactions on Internet Technology* 23.1 (2023), pp. 1–26.

[25] Feijie Wu et al. "Infinity battle: A glance at how blockchain techniques serve in a serverless gaming system". In: *Proceedings of the 28th ACM International Conference on Multimedia*. 2020, pp. 4559–4561.

[26] Mingli Wu et al. "A comprehensive survey of blockchain: From theory to IoT applications and beyond". In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 8114–8154.

[27] Jingxi Xu and Benjamin W Wah. "Consistent synchronization of action order with least noticeable delays in fast-paced multiplayer online games". In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 13.1 (2016), pp. 1–25.

[28] Amir Yahyavi, Kévin Huguenin, and Bettina Kemme. "Interest modeling in games: The case of dead reckoning". In: *Multimedia systems* 19 (2013), pp. 255–270.

[29] Amir Yahyavi and Bettina Kemme. "Peer-to-peer architectures for massively multiplayer online games: A survey". In: *ACM Computing Surveys (CSUR)* 46.1 (2013), pp. 1–51.

[30] Anatoly Yakovenko. "Solana: A new architecture for a high performance blockchain v0. 8.13". In: *Whitepaper* (2018).

[31] Ho Yin Yuen et al. "Proof-of-play: A novel consensus model for blockchain-based peer-to-peer gaming system". In: *Proceedings of the 2019 ACM interna-*

tional symposium on blockchain and secure critical infrastructure. 2019, pp. 19–28.

[32] Jusik Yun et al. "MMOG user participation based decentralized consensus scheme and proof of participation analysis on the bryllite blockchain system". In: *KSII Transactions on Internet and Information Systems (TIIS)* 13.8 (2019), pp. 4093–4107.

Appendix

A Massively Multiplayer Online Game and Blockchains

Massively Multiplayer Online Game latency varies based on genre and user requirements. While low latency is not necessarily crucial for turn-based games, it is essential for high-stakes scenarios and e-sports competitions to enhance player experience and audience engagement. Table 2 analyses latency based on Massively Multiplayer Online Game usage categories and criticality needs. Since latency is inadequate for measuring criticality, we introduce actions per minute as an alternative metric.

Comparing Massively Multiplayer Online Game latency with blockchain latency requires aligning metrics. For Massively Multiplayer Online Games, we use latency and Action Per Minute, while for blockchain, we use block confirmation time and transactions per second (TPS). Block confirmation times range from minutes to a minimum of 400 milliseconds with Solana. The transaction per second range can align with certain Massively Multiplayer Online Game genres, such as first-person shooters with low transaction per second blockchain and others with high transaction per second blockchain.

Game categorize	Latency (ms)
LAN-local	0-10
Professional ESport	10-15
Amateur ESport	15-30
Leisure	30-50
LAG	50+

Game Genre	Action Per Minute
Fighting	150-180
Multiplayer online battle	200-300
Real time strategy	300-350+
First person shooters	25

Table 2: Latency and Action Per Minute metrics

Narrative adventure games are promising candidates for blockchain-based Peer-to-peer versions due to their Action Per Minute and latency characteristics, aligning well with blockchain architecture's flow of time 3.

blockchain	consensus algorithm	Transaction Per Seconds	number # confirmation block	validation time minute	finality or fork	Weight of block in Kilobyte
Ethereum [23]	proof of stake	15	30	6	finale	3662,109375
Polygon [16]	proof of stake	38	128	5	finale	2441,40625
Solana [30]	proof of history	4700	372	400 milliseconds	fork	NaN
Binance [4]	NaN	NaN	NaN	NaN	NaN	NaN

Table 3: Comparison between Binance, Ethereum, Polygon, Solana

B Cheating in peer-to-peer game

In the columns, a classification proposed by Yahyavi et al. [29], which categorises attacks according to their nature.

- Interruption of information dissemination: confuse the player about the current stage of the game
- Illegal game action: A client can evade the physical laws of the game and unfairly change its state by manipulating the game state.
- Unauthorised access to information: exploiting information that should not be disclosed.

On the rows is a classification proposed by Webb et al. [21], which categorises the attack at the level at which it occurs.

- Game level: occurs within the game program without any modification or external influence.
- Application level: requires modification of the game executable from data files or running programs that read from and write to the game's memory while it is running.
- Protocol level: interfering with the packets sent and received by the game.
- Infrastructure level: modifying or interfering with the software (e.g. drivers) or hardware (e.g. network infrastructure) used by the game.

C Challenger Distributed Services

Overall service architecture is presented in Figure 3, while the description of all services is presented in Table 5.

C.1 Membership Service

Membership service is described in Figure 4, while a corresponding sequence diagram is presented in Figure 5. The user's Orchestration initiates the process by sending the id_client to the Membership service. Membership service then performs two tasks:

- ID Creation: Membership service generates a unique user ID (id_user) and game ID (id_game) based on the client's provided id_client.
- ID Verification: Membership service reads the blockchain to ensure that the generated id_user and id_game are unique and not already in use. If the IDs are unique, they are sent to the Publication service; otherwise, the request is discarded.

C.2 Conflict service

The conflict service is described in Figure 6, while a corresponding sequence diagram is presented in Figure 7. The user initiates a verification process, leading to a call to Conflict service. Upon identifying the problem of multiple possible paths, Orchestration is invoked to obtain the user's perspective. The user can then choose between a manual selection and a choice proposed by the blockchain.

	Interruption information dissemination	Illegal game actions	Unauthorized information access
Game level		- Bug — Real money transaction	
Application level	- Fast rate cheat	- Client side code tampering — Map hack — Aim bots — Bots[13] — Reflex enhancers	- Information exposure — Invalid commands
Protocol level	- Replay cheat — Blind opponent — Suppress correct cheat — Time cheating [3] [8][9] — Collusion — Undo — Consistency cheat[26] [27]	- Spoofing	- Rate analysis
Infrastructure level	- Flooding — Escaping		- Sniffing — Information exposure

Table 4: Cheating Methods Summary Table

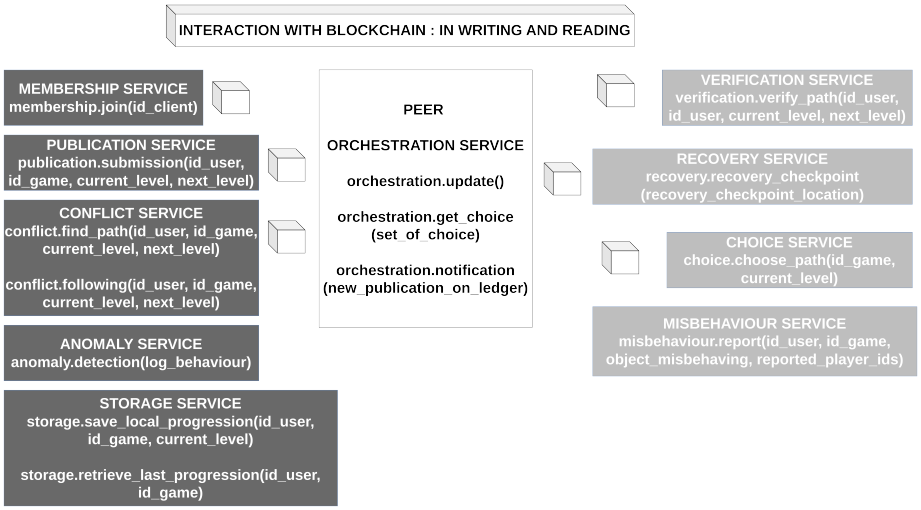

Fig. 3: Challenger Services Architecture

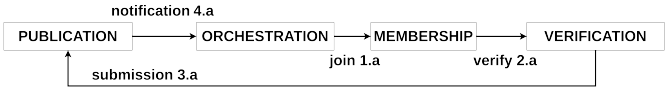

Fig. 4: Membership Service

SERVICE	Application Programming Interface	INPUT	OUTPUT	INTERACTION IN READING	INTERACTION IN WRITING
Membership	Join : first connexion to generate(id_user, id_game) based on id_client of client	id_client	(id_user, id_game) and deliver submission (id_user, id_client) to Publication service	Compare id_user and id_client with in the players IDs already Submitted on the blockchain. Verify player IDs are unique.	
Publication	submission : write the level up from current level to next_level on the blockchain as a proof of level change	(id_user, id_game, current_level, next_level)	deliver Notification (new information available on ledger) to Orchestration service		write the level up from current_level to next_level
Conflict	- Find_path : there is a problem with the progression of the game state of the user. Find the type of problem and send it to the right service	Conflict is called by Verification service using delivering Find_path (id_user, id_game, current_level, next_level)	Find_path return case : - deliver location_checkpoint_recovery to Recovery service - deliver Choice.Choose_path (current_level) to Choice service - deliver get_peer_choice(CHOICE_TO_MAKE) to Orchestration service	-To find location_checkpoint_recovery, read blockchain to search for submission where there is a call to Choose_path this means that there is at least a choice between 2 paths	
Verification	Verify_path : compare two game states and evaluate if the level up is legit or not.	(id_user, id_user, current_level, next_level)	call Publication Service OR call Conflict Service	Compare the level up between current_level and next_level to level up, find with in the blockchain for the same current_level	
Recovery	Recovery_checkpoint : recover game state request can be made by users or a call by Conflict service	recovery_checkpoint _location	recovered_level	Find submission linked to recovery_checkpoint_location and extract recovered_level	
Choice	Choose_path : randomly drawn a path that is not register yet in the blockchain	id_game, current_level	random_path_drawn	- Find all possible paths given current_level - List all paths taken in the blockchain by users for current_level - Consider {not explored yet path} {all paths given current_level} minus {paths with in blockchain} randomly drawn, a path from {paths not explored yet}	
Storage	-Save_local_progression : save locally game state of id_user on id_game - Recover_local_progression : recover locally last game state saved by id_user on id_game	- (id_user, id_game, current_level) - (id_user, id_game)	- Notification (game_state_save) - last_game_state		
Misbehaviour	- Report : create a report with player IDs object of misbehaving and reported player IDs	(id_user, id_game, object_misbehaving, reported_player_IDs)	- log_behaviour		
Anomaly	- Detection : filter behaviour to determine cheating behaviour from other users behaviour	log_behaviour	log_anomaly		

Table 5: Challenger Services Summary

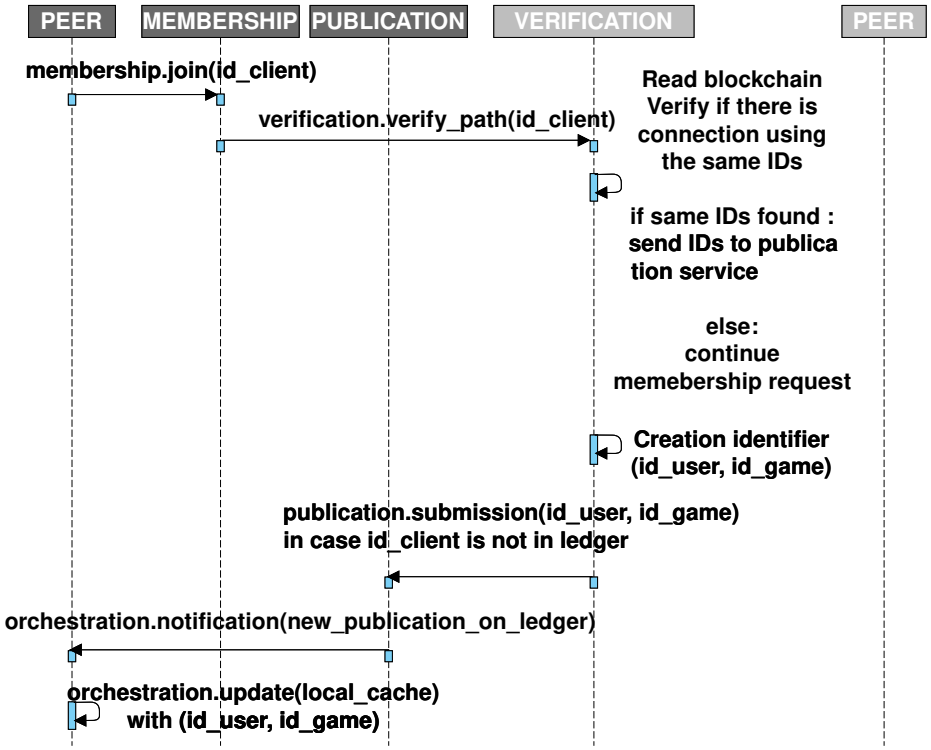

Fig. 5: Membership Service Sequence Diagram

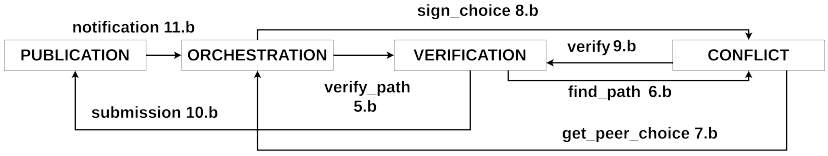

Fig. 6: Get client choice Architecture

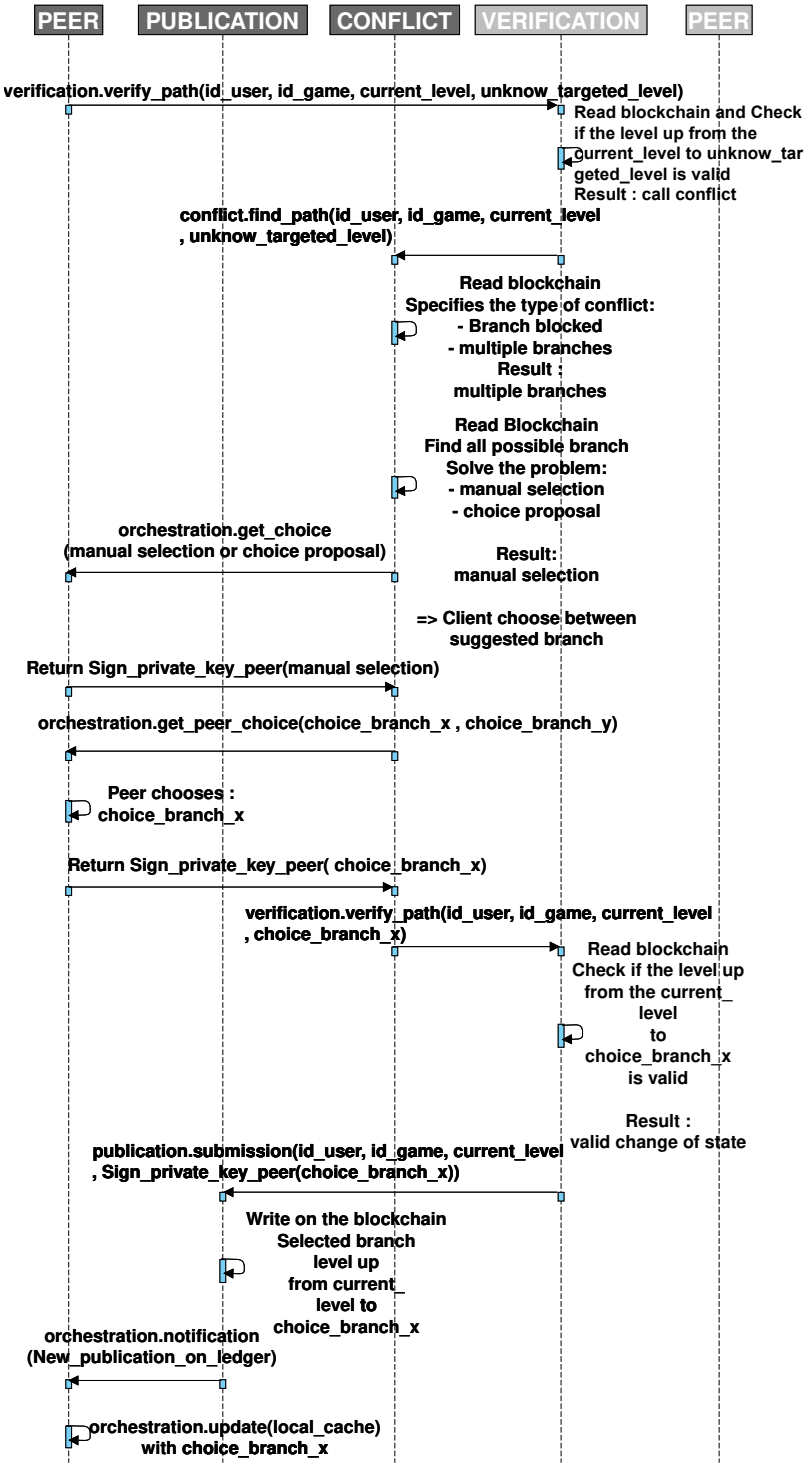

Fig. 7: Get client choice Sequence Diagram

C.3 Choice service

The choice service is described in Figure 8, while a corresponding sequence diagram is presented in Figure 9. A call to the Verification service initiates a subsequent call to the Conflict service due to path ambiguity. Conflict service identifies the potential paths a user can take from their position, referred to as SET OF PATH. When the Choice service is selected to decide the path, SET OF PATH is sent as input to the Choice service. A path is drawn according to the previously described process and is then submitted for updating by Orchestration in the user's local cache. The choice service Algorithm (Algorithm 5) takes the user's current level and all potential paths for that level as input. Instead of recalculating SET OF PATH, Choice uses it as input and reads the blockchain to extract paths already taken by users at this level. The difference between these two sets forms the set of potentially available and unexplored paths by other users.

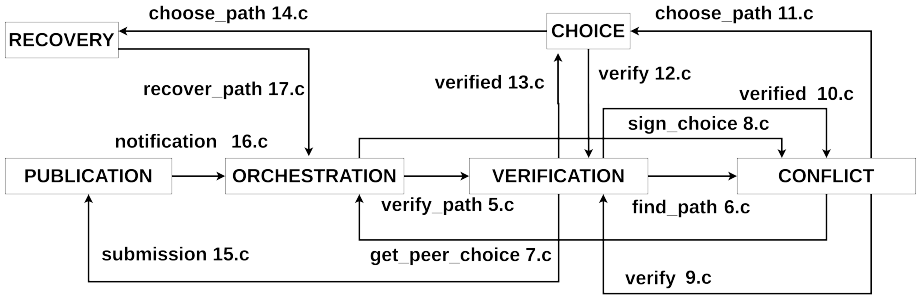

Fig. 8: Choice Service Architecture

C.4 Misbehaviour service

When requested by a user, the misbehaviour service is described in Figure 10, while a corresponding sequence diagram is presented in Figure 11. The Misbehaviour process begins with the Verification process, initiated by Conflict service. Conflict maintains a local variable, ALL GAME STATE STEP BY STEP, to respond to a verification process requested by a user. Conflict compares the request with ALL GAME STATE STEP BY STEP and reads the blockchain to detect redundancy or other malicious verification requests. If the Conflict's analysis matches any misbehaviour, a report is generated and collected in the LOG behaviour. The misbehaviour process stops here, as the filtering phase is part of the Anomaly service (Algorithm 9).

When requested by the system, the misbehaviour service is described in Figure 12, while a corresponding sequence diagram is presented in Figure 13.

The Anomaly filtering process targets the Misbehaviour service from the start when reports are received from users. This sequence diagram assumes that services are correct and users are altruistic. In the case where a user reports directly to another user for

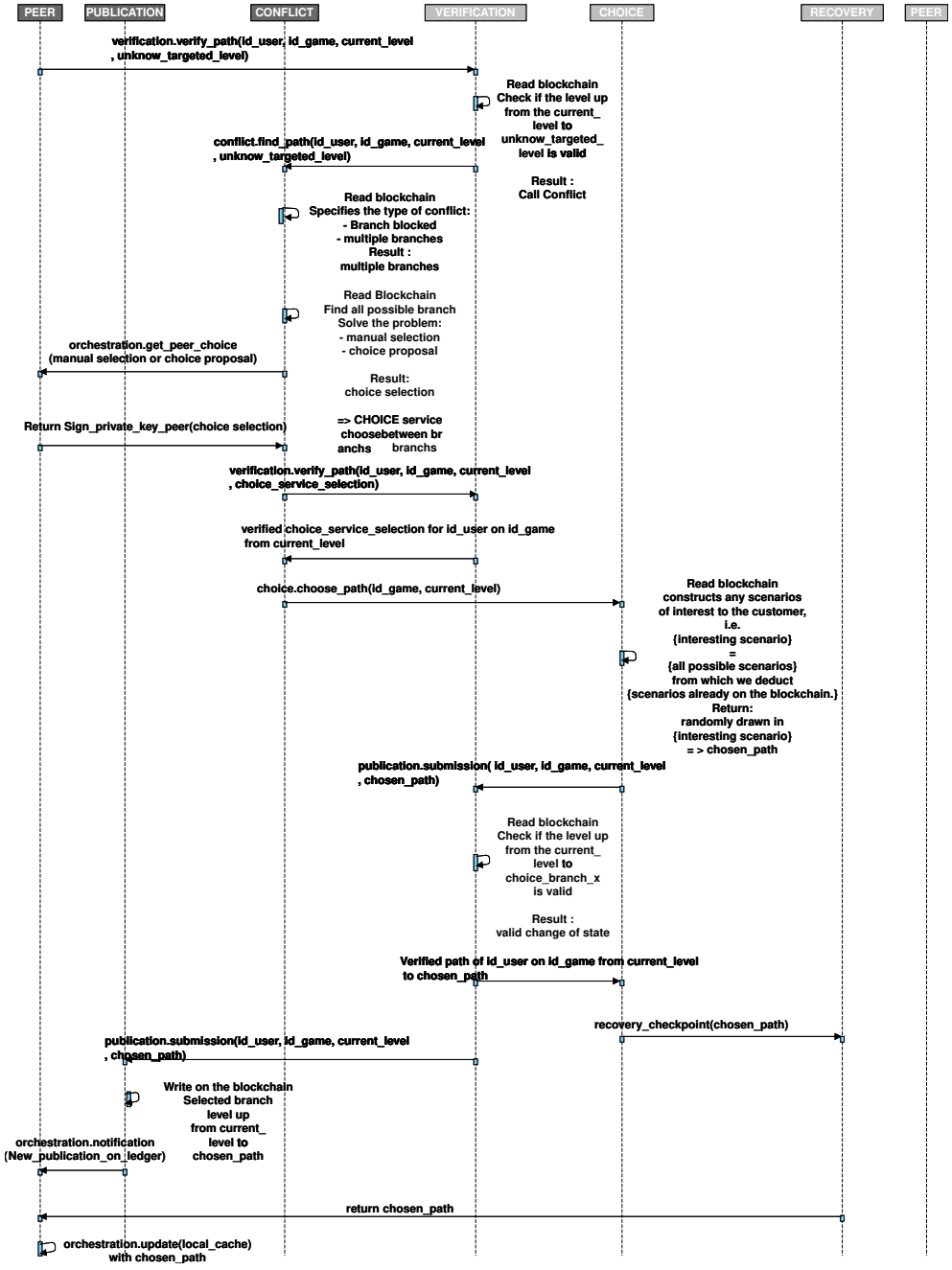

Fig. 9: Choice Service Sequence Diagram

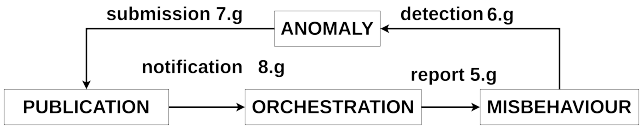

Fig. 10: Request Misbehaviour Architecture Service by user

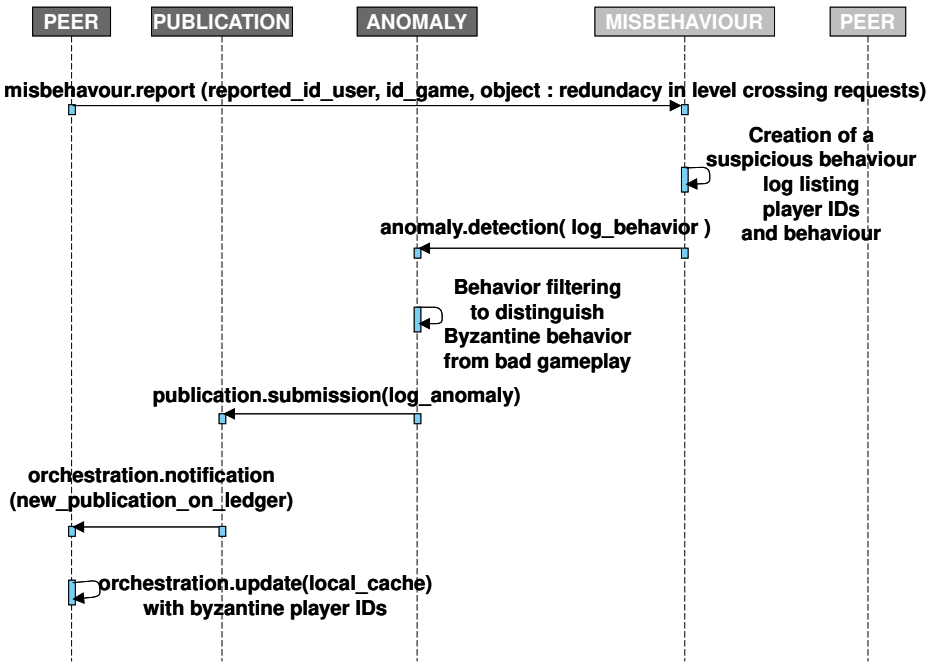

Fig. 11: Request Misbehaviour Sequence Diagram Service by user

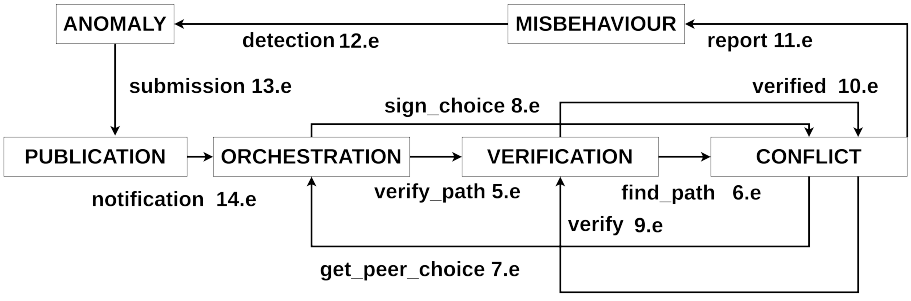

Fig. 12: Request Misbehaviour Service by Conflict Service Architecture

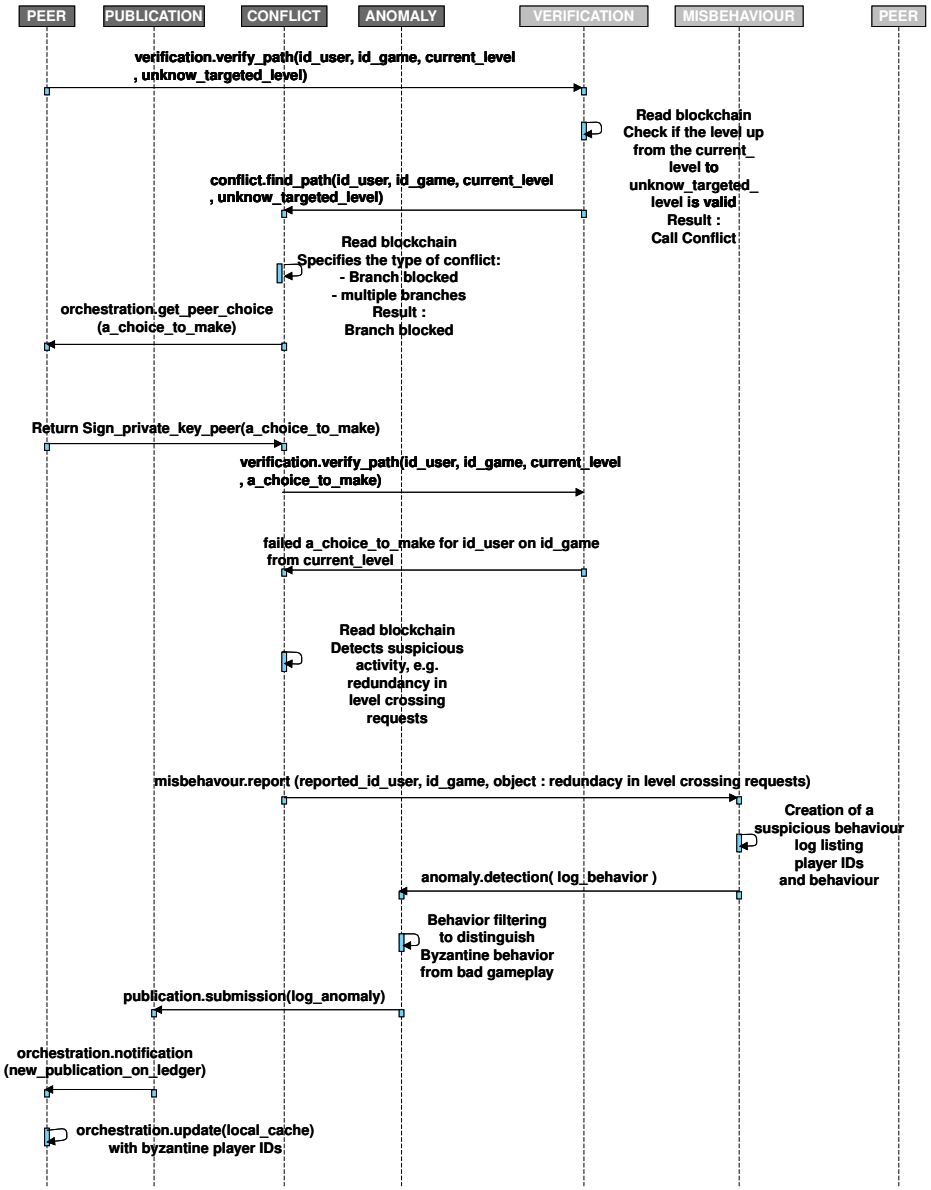

Fig. 13: Request Misbehaviour Service by Conflict Service Sequence Diagram

misbehaviour, there is a problem of trust that needs to be solved. For instance, from the point of view of the Anomaly service, there is no way to identify a legitimate report made by a cheating user and a suspicious report made by an altruist user.

C.5 Recovery service

When requested by the system, the recovery service is described in Figure 14, while a corresponding sequence diagram is presented in Figure 15.

The Recovery process is initiated by a call to Verification, which leads to a call to Conflict because the current level has no path (i.e., it is a dead end). Conflict reads the blockchain to find a previous game state with multiple paths. More specifically, it searches for a submission indicating that a user had to decide between several paths. This submission is registered as a location recovery and sent to the Recovery service. Recovery then reads the blockchain to find the location recovery location and sends the recovered game state to the user. Simultaneously, Publication is called to submit the location recovery, allowing the Application to update its local cache with the location recovery, which is useful if the user is blocked again and needs to recover the game state.

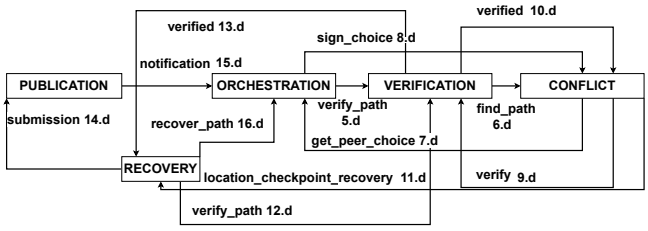

Fig. 14: Request Recovery service by Conflict Service Architecture

C.6 Orchestration service

Orchestration service is the client-side service that users interact with directly and transparently. Orchestration summarizes the interaction between him and the other services, as depicted in Figure 2. Interaction involves the following services:

- Membership service: Join, to create player IDs; Notification, to register player IDs
- Verification service: To verify level-up from two states of the game
- Publication service: Notification, to update new information on the ledger
- Conflict service
  - Get a choice, to choose between manual selection and Choice proposal
  - In case of manual selection, sign with a private key to select a path for the current level

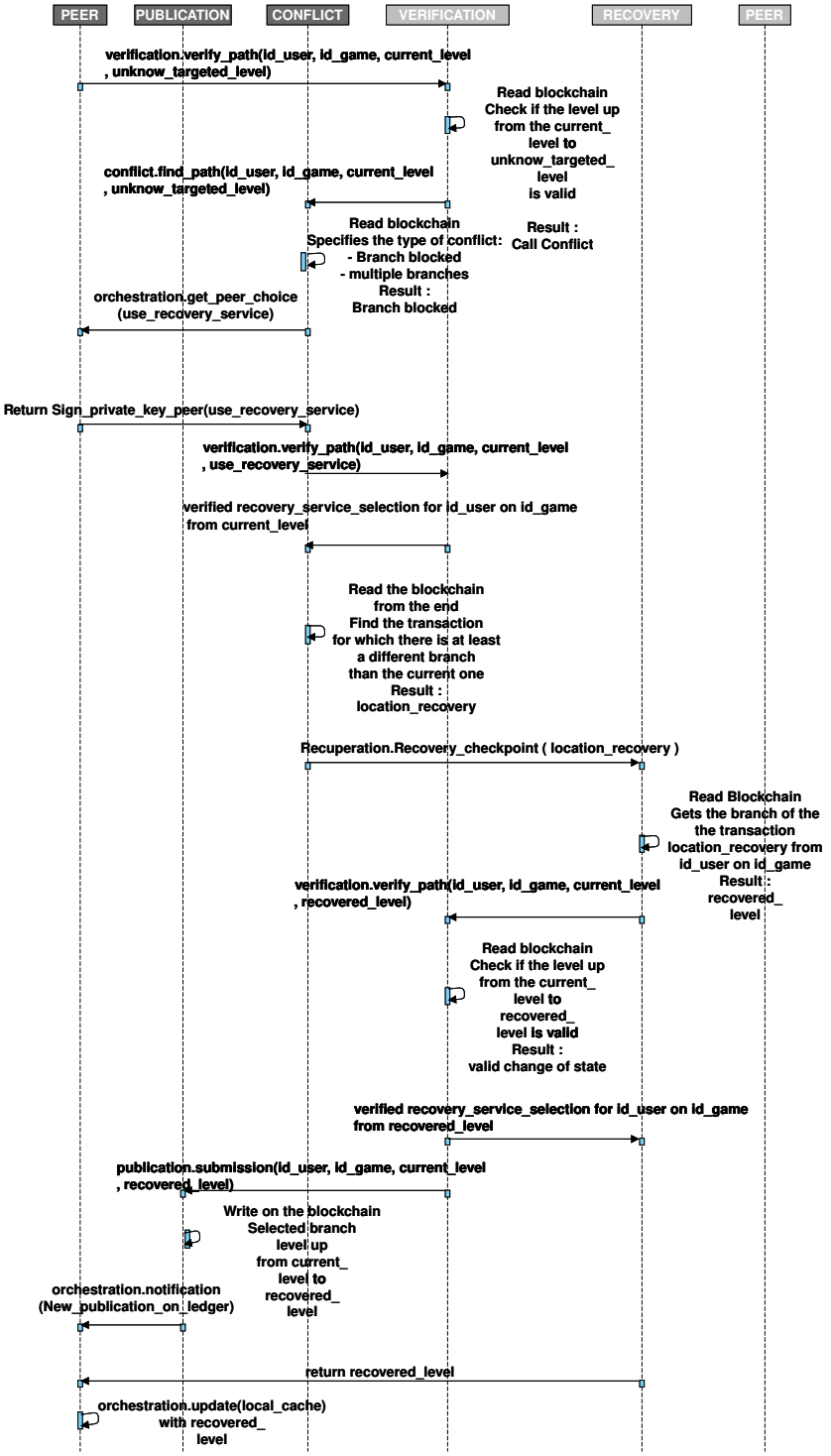

Fig. 15: Request Recovery Service by Conflict Service Sequence Diagram

- Misbehaviour service, to report misbehaviour
- Recovery service
  - user recovery: to recover the current level and start from scratch
  - Conflict Recovery, to recover the anterior level because user progression is blocked
- Storage: Save, to save local progression; Retrieve, to retrieve local progression

The pseudo-code for the orchestration service is provided as Algorithm 10.

Algorithm 10 Orchestration Service

```
1: Local variable :
2: LOCAL_CACHE local storage of application
3: Function :
4: - update (object):
5: take into account object and update LOCAL_CACHE if necessary
6:
7: Upon receiving Notification (player_IDs) from the Membership service do :
8:     update (LOCAL_CACHE)
9: enddo
10:
11: Upon receiving Notification (new_information_on_ledger) from the Publication service do :
12:     update (LOCAL_CACHE)
13: enddo
14:
15: Upon receiving Notification (log_anomaly) from the Publication service do :
16:     update (LOCAL_CACHE)
17: enddo
18:
19: Upon receiving Recovered_checkpoint from the Recovery service do :
20:     update (LOCAL_CACHE)
21: enddo
22:
23: Upon receiving get_peer_choice (id_user, id_game, current_level, CHOICE_TO_MAKE)
    from Conflict service do :
24:     deliver Sign_private_key_user (id_user, id_game, current_level, CHOICE_MADE) to
    Conflict service
25: enddo
26:
27: Upon receiving Notification (game_state_saved) from the Storage service do :
28:     update (LOCAL_CACHE)
29: enddo
30:
31: detection (log_behaviour) :
32: analysis of the misconduct to determine real Byzantine behaviour from stubborn or unwise
    game-play.
33: LOG_ANOMALY works as a reference for Byzantine attacks, the log helps to identify be-
    haviour that seems Byzantine.
34: LOG_ANOMALY must be constantly updated to predict potential Byzantine schemes.
```