Formalizing Rollback Netcodes for Robust and Real-Time Client-Server Architectures

Yérom-David Bromberg
Univ Rennes, Inria, CNRS, IRISA, France

Jérémie Decouchant
Delft University of Technology, The Netherlands

Manon Sourisseau
Univ Rennes, Inria, CNRS, IRISA, France

François Taïani
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

The rapid growth of the gaming industry has made netcodes (the part of an online game's source code that handles networking and synchronization) a critical component of the online multiplayer experience. Among various approaches, rollback netcodes have become a popular choice for real-time games due to their ability to enhance responsiveness and player immersion. However, despite their widespread adoption, these netcodes remain susceptible to subtle latency-based attacks that can be challenging to detect. Notably, while rollback netcodes play a critical role in the gaming industry and share similarities with synchronization mechanisms in distributed systems, they have received relatively limited attention in academic research.

In this work, we present a formal specification of rollback netcodes and identify key behavioral properties and requirements to strengthen their resilience against latency-based attacks that are prevalent in gaming, such as lag-switch and DDoS attacks. Our analysis allows us to explore the trade-offs between preserving immersive gameplay and ensuring security. Our findings reveal that ideal immersion requires strict assumptions about network latency, which are unattainable in adversarial environments where message delays are inevitable.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Online Multiplayer Game, Rollback Netcode, Network Security, Latency attacks

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2025.11

Acknowledgements This work was partially supported by the French ANR project ByBloS (ANR-20-CE25-0002-01) devoted to the modular design of building blocks for large-scale Byzantine-tolerant applications and by the École Normale Supérieure (ENS) of Rennes.

1 Introduction

Over the last few decades, the online gaming industry has been experiencing rapid growth, evolving from niche communities into a multi-billion-dollar industry [22]. Economically, online gaming has created new revenue streams through subscription models, in-game purchases, and advertising. Additionally, it has spurred the development of sophisticated gaming platforms and infrastructures, including cloud gaming services and robust networking technologies [14]. As online gaming continues to evolve, it remains at the forefront of innovation, driving both economic growth and technological progress in the digital entertainment sector [23].

A *Netcode* [24] refers to the intricate system that governs the communication and synchronization between players' devices and a game server in online multiplayer environments. It encompasses various algorithms and protocols designed ensure smooth gameplay, and

maintain fairness among players despite potential differences in network conditions. Essentially, a netcode is responsible for handling player inputs (players' actions), transmitting them to the server, processing them alongside inputs from other players, and updating the game state across all connected devices in real-time.

A defining feature of popular online games is their real-time nature, which not only enables a new kind of social interaction but also supports dynamic and immersive gameplay, enhancing the overall player experience. Unlike traditional delay-based netcodes, which often introduce noticeable input lag, causing a delay between a player's action and its on-screen response, a rollback netcode offers superior responsiveness by immediately updating the local game state based on the player's most recent input [15]. This is done under the optimistic assumption that the game state will remain consistent across all clients. To handle potential inconsistencies, the server continuously runs its own simulation of the game, serving as the authoritative source of truth. When a divergence is detected between a client's local state and the server's authoritative state, rollback netcodes "roll back" the game state to the last confirmed synchronized point (as sent by the server) and replay the player's recent actions. This mechanism effectively hides the effects of latency, resulting in a smoother, more responsive gameplay experience. A rollback netcode is especially beneficial for fast-paced, precision-focused games where split-second decisions matter, as it significantly reduces input delay while preserving synchronization across players, leading to more fluid and enjoyable online matches.

However, rollback netcodes can be targeted by attackers who may manipulate network latency to generate rollbacks that advantage them in the game. In particular, in this paper, we focus on the two most common latency-based attacks in online gaming: *Distributed Denial of Service* (DDoS) and *Lag Switch* [3, 7, 8]. In a DDoS attack, the cheater floods the network of the victim with an excessive amount of traffic, leading to bandwidth exhaustion and, thus, noticeable network issues impacting the game fluidity for the victim, giving the attacker an unfair advantage in the game. Conversely, in a Lag Switch attack, the attacker intentionally delays some of its network packets and performs some in-game actions, while momentarily appearing off-line to the server. By belatedly releasing the buffered packets corresponding to these actions, the attacker generates a rollback and gains an advantage. A video of each attack, performed on the Fortnite game, is available online [13]. One crucial aspect of these two attacks is their dual nature. If the attacks are well performed, the server cannot distinguish between network packets arriving late due to an attacker executing a Lag Switch attack or a victim suffering a DDoS attack. Moreover, these attacks are not only indistinguishable from each other but also from possible network latency changes, further complicating detection and mitigation efforts [3].

Interestingly, the academic literature on network architectures for online gaming remains limited [6, 16, 20, 26, 27], with particularly scarce work addressing netcode implementations [2, 3, 15]. Much of the available information comes instead from non-academic sources such as developer blogs [12, 17], open-source game engines [10], and technical articles from game companies [1, 19]. These resources provide valuable insights into the practical implementation and trade-offs of different netcode systems, and make it clear that *player immersion* has been the primary design focus of game developers, well ahead of safety and fairness.

In this paper, we formally investigate the in-game trade-off between immersion and safety. To do so, we propose the first property-based model of rollback netcodes. This model covers existing rollback netcode mechanisms and captures their intrinsic immersion requirements. Then, using this model, we study how existing attacks, such as DDoS and Lag Switch, can be avoided. Our results can find application to enhance the security of video games, as well as any applications that consider a real-time virtual world, such as metaverses.

Contributions. In this work, we make the following contributions:

- We introduce a model of rollback netcodes that captures their core behavior through two key properties.
- We define an *ideal immersion* property to formally represent the primary concern of game developers, using a formal construct that captures game semantics.
- Based on this model, we analyze the network latency requirements needed to uphold the immersion property, showing that it depends on low and bounded latencies, a condition that cannot be guaranteed in the presence of a message-delaying adversary.

This paper is organized as follows. Section 2 presents an in-depth description of rollback netcodes and discusses the related work. Section 3 describes our system model. Section 4 presents our model of rollback netcodes and their immersion requirements. Section 5 establishes the minimal network assumptions to be met to fulfill those requirements. Finally, Section 6 concludes this paper.

2 Background and State-of-the-Art

This section provides the necessary background on client-server rollback netcodes, with an highlight on latency-based attacks.

2.1 Client-Server Architecture and Netcodes

Clients/Server vs. P2P Architectures. Games based on netcodes are typically built on either a client-server or a peer-to-peer (P2P) architecture, each offering distinct advantages and trade-offs. In the client-server model, a central server manages the game state, processes player inputs, and ensures synchronization across all clients. This strong central authority enhances game integrity by mitigating cheating and maintaining consistency. However, the cost of maintaining a server infrastructure can be prohibitive for some game studios, leading them to prefer a P2P strategy. In contrast, P2P games often designate one player as the host, who acts as a de facto server to manage the game state. While this approach significantly reduces infrastructure costs, it introduces vulnerabilities such as increased susceptibility to cheating, security risks, and heavy dependence on the host's connection, computing power, and fairness. A few P2P games take a fully decentralized approach, distributing responsibilities across all players' devices. Although this can improve scalability and minimize infrastructure demands, it often exacerbates issues related to latency, security, and inconsistent gameplay [25]. Since most games relevant to our problem adopt either a client-server model or a P2P model with a host, we assume a client-server architecture in the following discussion.

Netcodes. Netcode is a broad term referring to the systems and algorithms that manage communication between players in online games, ensuring synchronization and responsiveness during gameplay. Two common types of netcodes are widely used: delay-based and rollback [4]. In delay-based netcodes, all players' inputs are delayed until they are received and processed by all participants, guaranteeing synchronization but often causing noticeable input lag under high latency conditions. Rollback netcodes, by contrast, minimize input latency by executing and rendering player actions immediately on the client's game state. This significantly enhances responsiveness – and by extension, player immersion – but introduces the risk of temporary desynchronization between clients [2, 15] The latency-based attacks we target in this article (such as DDoS and Lag Switch) are specific vulnerabilities associated with rollback netcodes.

2.2 Rollback Netcodes: an Immersion-Synchronization Trade-Off

We now provide a more technical overview of how rollback netcodes operate, along with the challenges they are designed to address.

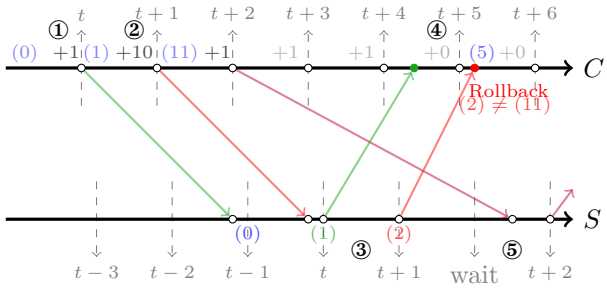

Figure 1 An overview of message transmission in a rollback netcode, between 1 client and the server. All the messages (arrows) are not represented. Messages with a similar color indicate that they refer to a similar game state.

Responsiveness. The main advantage of rollback netcodes is their responsiveness to player input. Since they are processed directly by clients and rendered to players (almost) immediately, without waiting for server approval, they are highly responsive. To explain how such a mechanism might lead to desynchronization between clients and servers, we must first delve into the technical details of a typical rollback netcode implementation.

Games operate on a system of ticks, a discrete unit of time during which the game processes updates such as player inputs, physics calculations, and AI behavior. These ticks occur at a fixed rate, measured in ticks per second (TPS), and act as the game's central clock. For example, a game running at 60 TPS updates the game every 16.67 milliseconds. A higher tick rate offers a smoother gameplay but requires greater computational resources. In a client-server architecture, the server ensures synchronization by standardizing the tick rate across all clients or by allowing varied tick rates but mapping each tick to standardized *moves* shared between clients. This approach accommodates clients with differing computational power without limiting the faster ones [19]. Crucially, clients send their inputs to the server at the same frequency, and the server manages synchronization. For instance, when the server receives inputs from client A for tick #354, it understands how this aligns with tick #287 of client B, whether or not those inputs have already been received. For the sake of simplicity, we consider in the following, without loss of generality, that the ticks of all clients are synchronized (e.g., Client A #200 ticks matches with Client B #200 ticks).

The server simulates a few ticks behind the clients to handle latency between clients and the server. When a player performs an input, it is immediately processed and rendered locally on the client's next tick. At the same time, the client sends this input to the server. The server integrates the received input into its simulation for the corresponding tick and later sends back its authoritative version of the game state to the client. If the server's state matches the client's local simulation, no correction is needed. However, if a significant discrepancy is detected, the client must acknowledge that its simulation for the affected tick was incorrect and re-simulate the current game state using the corrected data from the server.

Figure 1 illustrates an example of a flow of messages between one client and the server: The client (C) sends its input to the server (S) every tick. The inputs allowed are $\{-1, 0, +1\}$. ① At t, the player is in state (0) (for both client and server) and plays the input $+1$ for the

tick t, which locally changes its state from (0) to (1). The message carrying this input arrives at the server before the server reaches the t state. The server accepts the input and changes at t the authoritative state of the game and sends this state to the client. ② Between t and $t+1$, the client plays a wrong input (+10) that changes its local state to (11). ③When the server reaches the tick $t+1$ after receiving the wrong input, it refuses this input by computing an in-between value (2). Note that how the server refuses a value is a developer's choice (it could also have stayed in state (1), for instance). ④ When this corrected state reaches the client (just after its $t+5$), the client rolls back by correcting its state of $t+1$ by (2) and replays the inputs it has played by then $((2)+1+1+1+0)$, computing a new state (5). ⑤Then, due to a latency issue, the input of $t+2$ sent by the client arrives after the server reached its $t+2$ state, leading the server to wait (here one tick) to simulate its $t+2$ state, increasing the delay between the client's simulation and the server's simulation.

To compute its authoritative state, the server waits to receive inputs from all clients. This makes the server's simulation delay highly sensitive to network latency. In most games, there is a maximum delay threshold that defines how long the server will wait. If the delay reaches this threshold and some inputs are still missing, the server predicts them, often by assuming that the new input is the same as the previous one, and proceeds to simulate the game using this predicted data. Importantly, these predicted inputs are treated as authoritative, as they are used to update the official game state.

The further behind the server is in its simulation, the greater the risk of desynchronization between the server and clients, leading to an increased likelihood of rollbacks on the client side. The degree of delay in the server's simulation depends on the quality of the connection with each client: with a fast and stable connection, the server stays only a few ticks behind; with slower or unstable connections, it delays simulation further, up to the maximum threshold. This threshold varies significantly across games, even within the same genre, and can range from a few milliseconds to several seconds. The implications of choosing this threshold are discussed in more detail in Section 5.3.

Desynchronization in multiplayer environments. An essential characteristic of rollback netcode is that each client perceives other players' actions exclusively through the authoritative state provided by the server. Consequently, a client observes its own actions ahead of the server (which runs a few ticks behind) and perceives opponents' actions with additional delay, since their inputs must first transit through the server. This results in a persistent desynchronization among clients: no two clients ever render exactly the same game state. In this sense, the problem strongly diverges from the classical State Machine Replication problem [5], as consensus among states is never reached. Moreover, to preserve immersion, the server strives to accept actions as soon as they are valid in the client's local view. This approach, unfortunately, might lead to conflicts in case of incompatible concurrent updates, and cause rollbacks on clients.

Figure 2 illustrates such a semantic desynchronization in an online shooting game with two players (C^1 and C^2) at tick t. In C^1's view, actions are observed in advance of the server state, so C^2 appears in an outdated position and C^1 is allowed to shoot. In contrast, C^2's local view places itself ahead of the server state, safely hidden behind an obstacle, and thus out of C^1's line of fire. Importantly, the server validates C^1's shot, since it is consistent with C^1's perceived game state. Without this mechanism, players would need to aim at where opponents will be in the server's state, a latency-dependent guess, breaking immersion, as shooting directly at the target would no longer be reliable.

This divergence of views is intrinsic to rollback netcode. In practice, under normal latency conditions, such semantic desynchronization remains imperceptible to human players. However, adversarial conditions such as DDoS or LagSwitch attacks can amplify desynchronization, exploiting the immersion requirement and producing observable semantic conflicts, thereby granting an advantage to attackers. Consequently, in the remainder of this paper, the term *immersion* is used to denote the formal notion of semantic consistency among clients, particularly with respect to avoiding rollbacks that could compromise this consistency.

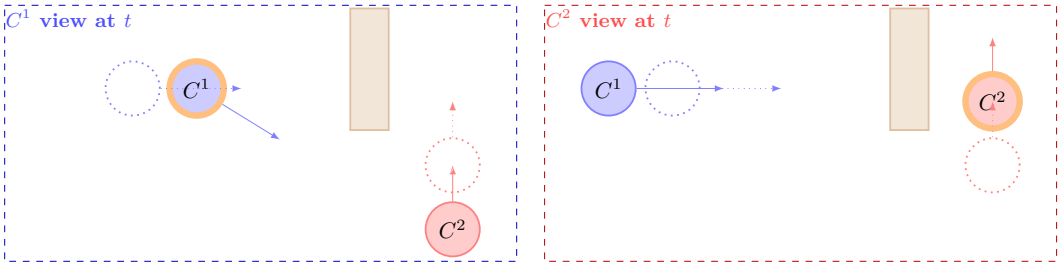

Figure 2 Semantic conflict due to desynchronization in a shooting game. The blue dotted rectangle indicates player 1's view at tick t, and the red dotted rectangle indicates player 2's view at the same tick. The brown rectangle denotes an obstacle that blocks line-of-sight. Solid circles represent current player positions, dotted circles indicate either the previous position of the focal player or the next position of the opponent, and the yellow circle shows the server's authoritative position of the player at tick t.

2.3 Immersive gameplay leads to latency attacks

Malicious clients can exploit those mechanisms to gain unfair advantages in the game. In the following, we highlight two undetectable latency attacks: the Lag-Switch attack and the DDoS attack [14].

Lag-Switch. The Lag-Switch attack exploits the immersion requirement in the following way: the attacker uses a physical or software switch to momentarily cut the connection between its client and the server at will. From the server's perspective, as long as this interruption is brief (depending on the max-delay threshold), it appears indistinguishable from a temporary spike in latency. During this disconnection, the attacker can freely perform special actions without interference since it no longer receives updates from the server about other players and thus perceives other players as inactive. Meanwhile, other clients remain unaware of the attacker's actions until the connection is restored.

Once the attacker reconnects and the delayed packets are delivered to the server, these actions are accepted since they are valid in the attacker's view and incorporated into the authoritative game state that the server sends to all clients. This leads to rollbacks on other clients, often breaking immersion.

DDoS. A Distributed Denial-of-Service (DDoS) attack is a coordinated effort to disrupt the normal operation of a targeted server, service, or network by overwhelming it with a massive volume of traffic originating from multiple sources [18, 21, 28]. Often part of a botnet composed of compromised devices, these sources generate traffic at a scale that makes it difficult to distinguish legitimate requests from malicious ones, thereby complicating mitigation efforts. In the context of online gaming, DDoS attacks can be performed by an

attacker to cut/slow down the latency between a victim client and the server. During the DDoS, the attacker can perform actions that make it gain some advantages while being sure that the victim cannot perform counteractions since the victim doesn't see its actions due to high delay/loss of the server's authoritative states.

If well-performed, both attacks are indistinguishable from each other and/or undetectable from natural latency jitter, because of natural network latency variations.

3 System Model

The previous section introduced the concepts behind rollback netcodes. In practice, implementations of rollback netcodes can somewhat diverge (through optimization, for instance) from our description. However, the abstracted mechanisms we described appeared common to all resources we identified about rollback netcodes. Responsiveness or immersion goals are also always referred to as the top priority of game designers. To enable formal reasoning about rollback netcodes, we abstract and formalize these mechanisms in a way that remains general enough to encompass a wide range of games and implementations. Before presenting our formal model of rollback netcodes, we first outline our system assumptions and introduce the associated formal notations.

We assume a set $\Theta = \{C^1, C^2, ..., C^n\}$ of $n \geq 2$ processes modeling the clients, and an additional process S modeling the server. Thus, the system contains a total of $n+1$ processes. The number of processes is known to the server, but is not supposed to be known by the clients. All processes are connected by a communication network, such that the server can communicate directly with each client, and each client communicates only with the server. We assume that the communication channels are reliable (i.e., messages are not lost). We assume that each process knows a common discrete unit of time called *tick* (for instance, one tick corresponds to 16 ms), and sends a message following a common tick rate. As explained in Section 2.2, we assume that the server can synchronize the tick numbers. For each client-server communication channels, we note λ_t^k the *ground truth* latency between the client k and the server at tick t, such that if client k sends a message to the server at tick t, then the server receives it at tick $t + \lambda_t^k$ [1] (and inversely with the server toward client k). This ground truth latency does not have to be known by the processes and is not supposed to be bounded.

On each process, there is a replica of the game state. Formally, we note S_t the replica of the server at tick t, and we note C_t^k, the replica of the client k at tick t. We suppose that there exists an initial game state, noted c_0, such that, for each k, $S_0 = C_0^k = c_0$. For each tick and each client, we suppose that the client knows a family of inputs performed by the player. We note $(I_t^k)_t$ this family, such that I_t^k corresponds to the player k's actions aggregated during the last tick. Table 1 summarizes the notations used in this paper.

4 Capturing Rollback Netcodes into Properties

In this section, we introduce a property-based model designed to capture the fundamental mechanisms of rollback netcodes, along with its core requirements.

We first present two properties capturing how the processes update their local replicas of the game state. The first one captures how clients update their local replicas depending on the last authoritative state received from the server, the latency, and the player's last inputs. The second one captures how the server updates its local replicas, depending on latency and the last received clients' inputs.

[1] Latency is usually measured in time units, and can thus be easily converted in tick units.

Table 1 Summary of notations.

Notation	Meaning
n	Number of clients
t	Ticks
C^k	Client $k \in [1, n]$
C_t^k	Replica's state of C^k at tick t
S_t	Replica's state of S at tick t
λ_t^k	Latency between C^k and S at tick t
U_c	Update function for clients
U_s	Update function for the server
δ	Semantic distance function
d	Size of states
η	Semantic threshold

Then, we model the immersion requirement by introducing a *semantic distance* between replicas, measuring how desynchronized two replicas are, especially about player's actions that can give an advantage in the game, leading to a property that limits this semantic distance between replicas.

The main purpose of the last immersion property is to evaluate the constraints it implies on the game semantics and latency. Moreover, in the next section, we detail how it is a beginning step to make rollback netcodes more robust against latency-based attacks.

4.1 Capturing Fundamental Mechanisms

Update Functions. We introduce two *update* functions that aim to be as general as possible. Those functions compute the new game state depending on the player's actions. The two functions have the same purpose: computing the next game state on replicas. We note U_c the update function used by the clients, and U_s the update function used by the server (two functions are necessary, although they have the same purpose, because they do not have the same signature/domain definition). Crucially, both functions are deterministic: identical inputs always produce identical outputs.

The U_c update function takes as input a state, usually the last received from the server, and a subsequence of local input (can also be mathematically seen as a polynomial of the local inputs), usually corresponding to the last performed local input since the tick corresponding to the state of the first parameter. The output of U_c is then a new state.

Similarly, the U_s function takes as input the last state computed by the server and n inputs coming from the n clients for the corresponding tick. The U_s function returns a new state, a server's authoritative state.

Time Functions. We introduce a simple time function that associates a replica state C_t^k or S_t to a tick, corresponding to which tick this state is computed. For instance, $T(C_5^2) = 5$ means that client 2 computed the state C_5^2 at tick $t = 5$.

Clients Property. The clients update their state at each tick and compute the new state depending on the last received state from the server while applying the last inputs performed by the player. This leads to the following property:

▶ **Property 1** (Validity – Client). *On the client side, the computation of the state at tick t depends on the last received state from the server, and the performed inputs up to tick t. It is moreover computed at tick t. More formally, for each $t \in \mathbb{N}^*$, for each $k \in [1, n]$,*

$$C_t^k = U_c(S_{t-\lambda_t^k}, (I^k_{t-\lambda_t^k+j})_{j\in[1,\lambda_t^k]})$$

$$T(C_t^k) = t.$$

This property characterizes the behavior of clients in rollback netcodes. In particular, rollback events are captured by the fact that if the server computes a game state that differs from the one locally computed by a client, specifically regarding the client's own player, then the client's subsequent computation of the game state will also differ from the case where both states initially matched. This divergence reflects the client's need to re-simulate its local state based on the authoritative data received from the server.

Server Property. The server updates its state as soon as it receives all the inputs necessary to compute it, depending on its last state. This led to the following property,

▶ **Property 2** (Validity – Server). *On the server side, the computation of the state at tick t depends of the previous state, and all the inputs performed by the clients during this tick t. It is computed as soon as it receives all inputs. More formally:*

$$S_t = U_s(S_{t-1}, (I_t^k)_k)$$

$$T(S_t) = t + \max_{k\in[1,n]}(\lambda_t^k).$$

Note that, since the clients compute their state C_t^k at tick t, they all send their input I_t^k at tick t. Thus, the server can compute its S_t state when the last input arrives, corresponding to the maximum latency of the clients at tick t.

Figure 3 illustrates these two properties.

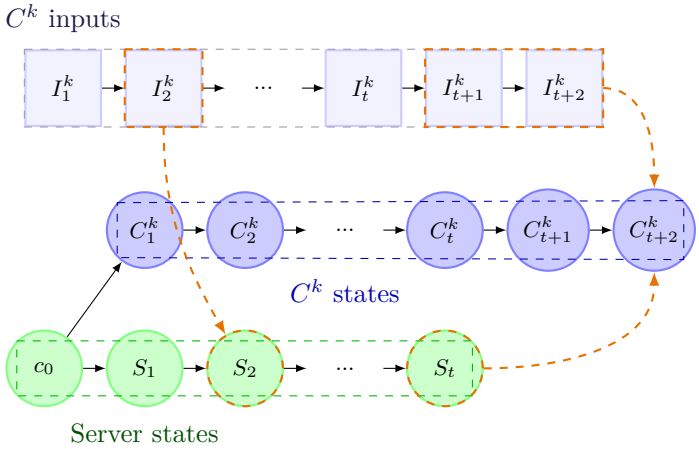

Figure 3 Scheme of updating replicas through ticks, illustrating validity properties. The server states correspond to the sequence of replica states (from c_0 to S_t). Same with C^k states, for client k. The sequence of inputs of clients k is also represented. The orange arrows scheme from within states/inputs, updates of states are computed, illustrating the two validity properties.

It is important to emphasize that this model abstracts away several optimizations typically employed in real-world implementations of rollback netcode. In practice, clients do not always recompute the entire game state from the last authoritative server update: if the locally predicted state coincides with the server state, recomputation might be partially skipped, and the client simply continues its simulation without rollback. Similarly, various caching and incremental update mechanisms are used to reduce the computational cost of re-simulations. These implementation details, while crucial for efficiency and playability, are deliberately omitted here in order to focus on capturing the fundamental mechanisms and correctness properties of rollback netcodes.

4.2 Capturing the Immersion Requirement

While validity properties aim to capture the practical behavior of rollback netcodes as implemented in modern games, we introduce here an additional property that reflects the idealized immersion goal often sought by game designers yet rarely fully achieved in practice. The purpose of this property is to formally specify the conditions, whether in terms of network performance or update functions (i.e., game semantics), that would be required to achieve seamless immersion within rollback-based architectures. The following section also explores how this property relates to known latency-based attacks on rollback netcodes.

Rollback events can introduce visual discontinuities in the player's rendering of the game, and such discontinuities must avoid producing a sense of unfairness or breaking immersion. Ideally, when a client tackles a rollback, the resulting visual outcome should not appear implausible or unnatural to the player. However, what constitutes an implausible or strange in-game action is inherently tied to the semantics of the game. It is, therefore, *necessary* to introduce a formal construct capable of capturing the game's semantic logic.

To do so, we introduce a *semantic distance* function, denoted δ, that measures how far apart two game states are from a semantic perspective. We assume that the virtual world, described by states, can be represented as a d-dimensional vector, where each dimension corresponds to a relevant game parameter (e.g., player position, health points, active abilities, environment objects). On such vectors, we define a weighted norm in which critical events are assigned higher weights, while frequent or less impactful actions receive lower weights. In this way, δ emphasizes discrepancies that are most likely to alter gameplay outcomes or player perception. Figure 4 illustrates this abstraction. The assignment of weights is inherently game-dependent, since what constitutes a "critical" action varies across genres. For instance, in a fighting game, the activation of a special move or a successful hit should carry a higher weight than background animations or minor positional adjustments. Thus, we consider that the choice of weights falls under the responsibility of game developers, who are best positioned to capture the semantics of their own game.

Based on this norm, we can then compute a semantic distance between two states corresponding to the same tick. To bound acceptable divergence, we introduce a *semantic threshold* η, which defines the maximum tolerable desynchronization across clients. Crucially, the weights must be calibrated such that whenever conflicting outcomes occur – e.g., a player perceives a successful hit while another perceives a miss – the resulting semantic distance exceeds η. This ensures that semantically inconsistent states can be detected and treated as conflicts. In the following, and for simplicity, we assume $\eta \geq 1$. The next section discusses how this parameter relates both to network latency and to the semantic granularity of the game, and how it can guide developers in choosing an appropriate value for η.

More formally, we assume a family of weight $(w_i)_{i\in[1,d]}$, and note $C_t^k[i]$ the i-th parameter of the state C_t^k. We then define the semantic distance δ as a weighted Euclidean distance:

$$\delta(s_1, s_2) = \sqrt{\sum_{i=1}^{d} w_i(s_1[i] - s_2[i])^2}$$

This formulation makes the role of weights explicit: large values of w_i increase the contribution of semantically critical parameters to the overall distance, while small values attenuate the effect of less significant differences.

Note that, specifically to the immersion requirement, is that what matters is what *the player sees on its screen.* Thus, if player A sees important actions that player B doesn't see, then having an important desynchronization on those actions will not break the immersion requirement. We thus introduce *projector* functions, noted $(p_t^k)_{k,t}$, that take as input a state and weight to zero the elements of the virtual world the player k doesn't see at tick t. For instance, on Figure 4, if Player 1 doesn't see Player 2 (then doesn't render it on the screen), at tick t, then the parameters of $p_t^1(C_t^1)$ describing Player 2 are weighted to 0.

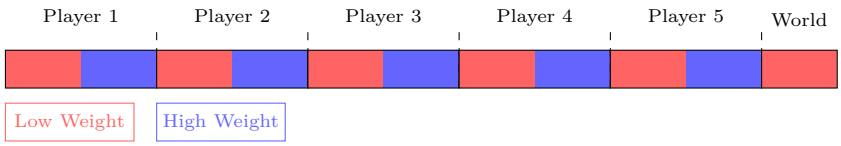

Figure 4 Illustration of the model of States, using a vector of $d = 11$ parameters, describing here five players' states and the world (map)'s state. For each client, a low weight is attributed to common action (such as movement, animation status, basic attack, ..., depending on the game's semantics), whereas a higher weight is attributed to special action (shooting, special attacks, casting spells, ...).

Thus, the immersion requirement can be described formally by the following property:

▶ **Property 3** (Immersion-Consistency). *All client states must be semantically close to the server states, for any ticks. More formally: For all $k \in [1, n]$, for all $t \in \mathbb{N}*$,*

$$\delta(p_t^k(S_t), p_t^k(C_t^k)) \leq \eta.$$

This property introduces limitations on how Clients' states can evolve depending on the Server's states, in terms of special actions, described by the semantic distance δ. In the next section, we study the impact of such a property on the system requirement and show how this property is linked with known latency-based attacks.

5 Implication of Consistency

Property 3 establishes a limitation to server states compared to clients' states, in terms of semantics distance. In the following, we first study what it implies for the clients' semantics compared to the Server's semantics locally, as well as what it implies for network latency. We then show how assuring this property is a shield against the latency-based attacks.

5.1 Implication of Game Semantics on Network Latency

In the following, we assume the three properties (Validity-Client/Server, Consistency) are ensured. We define a join operator $\oplus$ on projectors, such that, for two clients k_1, k_2, we have:

$$p_t^{k_1 \oplus k_2}(x_1, ..., x_d) = \Big(\min\big(p_t^{k_1}(x_1), p_t^{k_2}(x_1)\big), ..., \min\big(p_t^{k_1}(x_d), p_t^{k_2}(x_d)\big)\Big). \tag{1}$$

Intuitively, this operator creates a new projector that corresponds to what both k_1 and k_2 render.

▶ **Lemma 1.** *For any pair of clients, the semantic distance of their joint projection, for any tick, is always lower than 2η. More formally : For all $k_1, k_2 \in [1, n]$, for all $t \in \mathbb{N}^*$,*

$$\delta(p_t^{k_1 \oplus k_2}(C_t^{k_1}), p_t^{k_1 \oplus k_2}(C_t^{k_2})) < 2\eta. \tag{2}$$

Proof. Since we assume consistency, we have, for both k_1 and k_2:

$$\forall t, \delta(p_t^{k_1}(S_t), p_t^{k_1}(C_t^{k_1})) \leq \eta \wedge \delta(p_t^{k_2}(S_t), p_t^{k_2}(C_t^{k_2})) \leq \eta. \tag{3}$$

In particular, using the definition of the $\oplus$ operator, we have, for all t:

$$\begin{cases} \delta(p_t^{k_1 \oplus k_2}(S_t), p_t^{k_1 \oplus k_2}(C_t^{k_1})) & \leq \eta, \\ \delta(p_t^{k_1 \oplus k_2}(S_t), p_t^{k_1 \oplus k_2}(C_t^{k_2})) & \leq \eta. \end{cases} \tag{4}$$

Thus, by summing both terms, we have, for all t:

$$\delta(p_t^{k_1 \oplus k_2}(S_t), p_t^{k_1 \oplus k_2}(C_t^{k_1})) + \delta(p_t^{k_1 \oplus k_2}(S_t), p_t^{k_1 \oplus k_2}(C_t^{k_2})) \leq 2\eta. \tag{5}$$

Since δ is a geometrical distance, by using the triangle inequality:

$$\delta(p_t^{k_1 \oplus k_2}(C_t^{k_1}), p_t^{k_1 \oplus k_2}(C_t^{k_2})) \leq 2\eta. \tag{6}$$

Property 3 is a bound on semantic deviation between any client and the server. Naturally, Lemma 1 highlights a bound of semantic deviation between two clients.

▶ **Theorem 2.** *All the clients are semantically bound to the last server state received by all clients. More formally: For all $k \in [1, n]$, for all $t \in \mathbb{N}^*$,*

$$\forall j \in [0, \max_{k \in [1,n]}(\lambda_t^k) + \max_k(\lambda^k_{\max_k(\lambda_t^k)})], \delta(p_{t+j}^k(C_{t+j}^k), p_{t+j}^k(S_t)) \leq \eta. \tag{7}$$

Note that $\max_k(\lambda_t^k) + \max_k(\lambda^k_{\max_k(\lambda_t^k)})$ corresponds to the longest latency of Client-Server round-trip (longest rtt though the clients) at tick t.

Proof. Let's show that theorem is an equivalence of Property 3. The sense of the Theorem to the property is direct (for $j = 0$).

Let's suppose that it exists t and $j \in [O, \max_{k \in [1,n]}(\lambda_t^k) + \max_k(\lambda^k_{\max_k(\lambda_t^k)})]$, such that each client computes the state $t + j$, where, for a client k_1, the last received state from the server is S_t, while for a client k_2, the last received state from the server is S_{t+1} (i.e., $\lambda_{t+1}^{k_1} > x > \lambda_{t+1}^{k_2}$). It is thus possible to find two inputs sequences $(I_t^{k_1})_t, (I_t^{k_2})_t$, such that each client respect the immersion-consistency proporty, but violate Lemma 1, where $\delta(C_{t+j}^{k_2}, S_{t+1}) \leq \eta$, and $\delta(C_{t+j}^{k_1}, S_t) \leq \eta$, but $\delta(C_{t+j}^{k_1}, C_{t+j}^{k_2}) > 2\eta$. Such a situation is illustrated in Figure 5. To avoid such a situation, it is thus necessary to limit every client semantically to the last received state by every client. Since the state S_t is computed as time $T(S_t) = t + \max_k(\lambda_t^k)$, all client must be bounded up to $t + \max_k(\lambda_t^k) + \max_k(\lambda^k_{\max_k(\lambda_t^k)})$. ◀

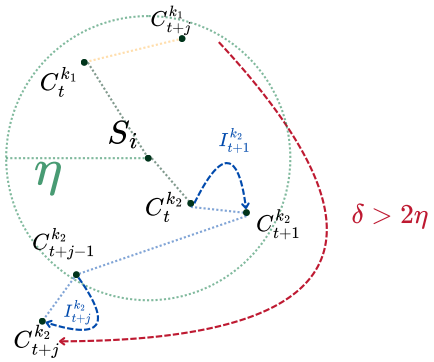

Figure 5 Scheme of proof of Theorem 2.

Theorem 2 is an equivalence of Property 3 (considering validity properties) and defines the requirements in terms of both latency and game semantics to ensure the immersion property. The main challenge highlighted here is that the evolution of a client's state is highly dependent on the latency of other clients.

For instance, Theorem 2 emphasizes that if a client A experiences high latency, then client B must perform actions that are semantically close to its previous actions, with the degree of constraint depending on the severity of client A's latency. In practice, enforcing such a requirement is highly challenging, as clients typically lack information about other clients' latencies.

Moreover, this could open the door to potential attacks: a malicious client could deliberately increase its latency to disrupt other players' gameplay, limiting their available actions.

A possible safeguard against this issue is to impose a maximum latency threshold, beyond which a timeout occurs. This approach is already used in some modern games [1, 19], where the server predicts a client's input if the latency exceeds an acceptable limit. Such a safeguard is discussed in the Latency-based Attack subsection.

▶ **Corollary 3.** *For all ticks, each client is locally semantically bound in its computation of its next state, as well as the server is. More formally: for all* $k \in [1, n]$*, for all* $t \in \mathbb{N}^*$*,*

$$\delta(C^k_t, C^k_{t+1}) \leq 2\eta \wedge \delta(S_t, S_{t+1}) \leq 2\eta. \tag{8}$$

Proof. We recall that for all ticks t and all clients k, we have $\lambda^k_t \geq 1$. Following Theorem 2, we thus have:

$$\delta(C^k_{t+1}, S_t) \leq \eta \wedge \delta(C_t, S_t) \leq \eta. \tag{9}$$

By summing the two terms, using the triangle inequality, we have:

$$\delta(C^k_t, C^k_{t+1}) \leq \delta(C^k_{t+1}, S_t) + \delta(C_t, S_t) \leq 2\eta. \tag{10}$$

Similarly, we have:

$$\delta(S_{t+1}, C^k_{t+1}) \leq \eta \wedge \delta(S_t, C^k_{t+1}) \leq \eta. \tag{11}$$

Thus, we have

$$\delta(S_t, S_{t+1}) \leq 2\eta, \tag{12}$$

concluding the proof. ◀

Property 3 induces Corollary 3, but the contrary is false. However, it interestingly shows how clients and server must be bounded locally, in terms of semantics. Such a result is discussed in more detail when characterizing the special actions in the next subsection.

5.2 Characterization of Special Actions

As previously described, special actions refer to critical player inputs or movements that, in our model, produce significant semantic changes, parameterized by η. In this section, we formalize and characterize such actions within the framework of our model.

Definition 4 defines formally special actions:

▶ **Definition 4** (Special Action). *We say that a client k performs a special action at tick t if there exist $x \in \mathbb{N}^*$ and $\alpha \in \mathbb{R}$ such that $\delta(C_t^k, C_{t+x}^k) = \alpha\eta$, with $\alpha > 2$. Without loss of generality, we assume x is the smallest integer satisfying this property.*

This subsection aims to analyze the relationship between x and α. Intuitively, the greater the value of x, the larger α tends to be, since a more substantial semantic shift requires a longer sequence of ticks to preserve the immersion requirement, in contrast to minor changes.

Building on Theorem 2, we can derive a lower bound on x depending on α. Let us define $f(t) = \max_k(\lambda_t^k) + \max_k(\lambda^k_{\max_k(\lambda_t^k)})$, representing the maximum round-trip latency among all clients at tick t. According to Theorem 2, if a client intends to perform a semantic change exceeding 2η, it must wait at least $f(t)$ ticks. This leads to the following lower bounds:

$$\begin{aligned} &\alpha \geq 2 \Rightarrow x > f(t), \\ &\alpha \geq 4 \Rightarrow x > f(t) + f\big(f(t)\big), \\ &\quad \ldots \end{aligned} \tag{13}$$

In the general case, for $u \in \mathbb{N}^*$, $\alpha \geq 2u \Rightarrow x > \sum_{j=1}^{u} f^j(t)$, where $f^j(t)$ denotes the j^{th} composition of f evaluated at t. In the special case where f is constant (i.e., $f(t) = f_0$ for some fixed upper bound on latency), this simplifies to $x > u \times f_0$.

This analysis highlights the significant impact of the parameter η on gameplay dynamics. A larger η implies a longer delay, dependent on all clients' latencies, before a special action can be completed. Consequently, in games where such actions are expected to be short in real-time, satisfying the immersion requirement imposes stringent constraints on latency. However, we argue that game developers can leverage visual feedback, such as animations or environmental cues, to create the illusion of rapid action execution, even if the underlying semantic change takes longer in the game state. Comparable design strategies are already widely adopted, particularly in the fighting game genre, where extended animation sequences are deliberately used to mask the perceptual impact of rollback corrections [9].

5.3 Link with Latency-Based Attacks

The immersion requirement, formalized by Property 3, acts as a defense mechanism against both Lag-Switch and Distributed Denial-of-Service (DDoS) attacks, since it prevents players from experiencing semantically diverging states during critical gameplay moments. By enforcing consistency in the perception of outcomes (e.g., whether an action succeeds or fails), the system ensures that no client can exploit temporary desynchronization to gain an unfair advantage. However, Theorem 2 highlights the cost of this guarantee: satisfying Property 3 relies on strong assumptions about both network latency and the semantic distance function introduced earlier. In particular, the theorem shows that the property cannot be maintained in models where adversaries are allowed to arbitrarily delay messages, as such adversaries directly control the effective latency observed by the system.

This exposes a core tension: immersion requires clients to act on game states that remain semantically consistent with their own view, yet achieving such consistency depends on fragile assumptions about bounded network delays. One mitigation is to impose an upper bound on admissible latency, discarding or delaying inputs once this limit is exceeded. However, this opens a new vulnerability: an attacker could inflate a victim client's latency to block this client's actions from the authoritative state. Conversely, lowering the threshold improves gameplay fluidity under normal conditions by reducing visible rollbacks, but also weakens resilience to adversarial or transient latency spikes.

Finally, we establish that clients must be aware of the latencies of other participants to safely perform certain semantically critical actions. In practice, this requirement is only achievable under the assumption of a known and enforceable latency threshold, which bounds the maximum divergence between clients. Otherwise, an adversary can arbitrarily increase delay and cause semantic conflicts that break immersion. This results in the impossibility of achieving both perfect immersion and reliability against message adversaries.

Such results resonate with the classical FLP impossibility of achieving consensus in asynchronous systems in the presence of process faults [11]. However, our contribution offers a different perspective, as it does not address consensus directly but rather the notion of immersion. While both immersion and consensus concern the ability of nodes to reach agreement on a common subject, immersion is a fundamentally distinct concept due to its semantic interpretation and its use of view projection. In this sense, we believe our results extend the understanding of coordination limits in asynchronous systems beyond the traditional scope of consensus.

6 Conclusion

In this paper, we presented the first formal model of rollback netcode architectures, capturing their core behavioral principles and the trade-off between safety and immersion in client–server distributed games. Our model is defined through two fundamental properties governing how the server and clients update their local copies of the game world. On top of this foundation, we introduced the immersion requirement, a semantic condition expressing when a consistent and engaging gameplay experience can be maintained. This formalization not only clarifies long-standing ambiguities in how rollback mechanisms interact with semantic consistency but also provides a rigorous basis for reasoning about desynchronization and latency-based attacks such as Lagswitch or DDoS. Our analysis demonstrates that maintaining ideal immersion implies a synchronous communication model with short, bounded rounds tied to the game's semantics, conditions that become unattainable under adversarial message delay. Beyond this theoretical limit, our formal framework offers a tool for system designers to reason about trade-offs between responsiveness, consistency, and resilience. Future work will explore alternatives to strict latency bounds, aiming to either develop better safeguards or relax the immersion requirement and analyze its resilience against desynchronization-based attacks. Furthermore, although some attacks may be theoretically undetectable, in practice they are executed by human players or require knowledge of real-time latencies, suggesting that a significant portion of real-world attacks leave detectable traces; designing mechanisms to identify such patterns represents a promising direction for future research.

References

1 GDC 2025. Overwatch gameplay architecture and netcode. URL: https://www.youtube.com/watch?v=W3aieHjyNvw.

2 Mamun Ahmed, Saha Reno, Md. Rifat Rahman, and Sarjil Hassan Rifat. Analysis of Netcode, Latency, and Packet-loss in Online Multiplayer Games. In *2022 International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, pages 1198–1202, 2022.

3 Iliès Benhabbour, Yérom-David Bromberg, Marc Dacier, Sven Dietrich, Rodrigo Miragaia Rodrigues, and Paulo Esteves-Verissimo. Attacks on tomorrow's virtual world. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 105–110, 2023.

4 Tony "Ponder" Cannon. Ggpo rollback networking sdk. URL: https://www.ggpo.net/.

5 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI 99*, New Orleans, LA, 1999. USENIX Association. URL: https://dl.acm.org/citation.cfm?id=296824.

6 Minyeop Choi, Gihyuk Ko, and Sang Kil Cha. BotScreen: Trust everybody, but cut the aimbots yourself. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 481–498, Anaheim, CA, 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/choi.

7 Mark Claypool and Kajal Claypool. Latency can kill: precision and deadline in online games. In *ACM SIGMM*, pages 215–222, 2010. doi:10.1145/1730836.1730863.

8 Mark Claypool, Tianhe Wang, and McIntyre Watts. A taxonomy for player actions with latency in network games. In *ACM NOSSDAV '15 Workshop*, pages 67–72, 2015. doi:10.1145/2736084.2736093.

9 Analysis: Why rollback netcode is better. URL: https://youtu.be/0NLe4IpdS1w?si=mE0CPVsy1tTyIvOh.

10 Unreal Engine. "how to implement rollback netcode on ue5?". URL: https://forums.unrealengine.com/t/how-to-implement-rollback-netcode-on-ue5/1175132/1.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, 1985. doi:10.1145/3149.214121.

12 URL: https://gabrielgambetta.com/client-server-game-architecture.html.

13 ls game-sec lab: Fd and ddos attacks on fortnite. URL: https://www.dropbox.com/scl/fo/k1gjem9akbc1io7vw8ysk/AM_Uza91qNa9DTlHV_noCy0?rlkey=blaa5db9mcmjg816r0q9l0yhg&e=2&st=jdusdade&dl=0.

14 Andreas Haeberlen, Linh Thi Xuan Phan, and Morgan McGuire. Metaverse as a service: Megascale social 3d on the cloud. In *ACM SoCC'23*, pages 298–307, 2023. doi:10.1145/3620678.3624662.

15 Alain Lioret, Lior Diler, Sami Dalil, and Marion Mota. Hybrid Prediction for Games' Rollback Netcode. In *ACM SIGGRAPH 2022 Posters*, pages 1–2, 2022. doi:10.1145/3532719.3543199.

16 Shengmei Liu, Xiaokun Xu, and Mark Claypool. A survey and taxonomy of latency compensation techniques for network computer games. *ACM Comput. Surv.*, 54, 2022. doi:10.1145/3519023.

17 maintanksoftware. "rollback networking in dragon saddle melee". URL: https://www.maintanksoftware.com/article/rollback2.html.

18 Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet denial of service: attack and defense mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR, 2004.

19 Riot. "peeking into valorant's netcode ". URL: https://technology.riotgames.com/news/peeking-valorants-netcode.

20 Rosslin John Robles, Sang-Soo Yeo, Young-Deuk Moon, Gilcheol Park, and Seoksoo Kim. Online Games and Security Issues. In *2008 Second International Conference on Future Generation Communication and Networking*, pages 145–148, Hainan, China, 2008. IEEE. doi:10.1109/FGCN.2008.199.

21 Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak, and Ali A. Ghorbani. Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy. In *2019 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8, CHENNAI, India, 2019. IEEE. doi:10.1109/CCST.2019.8888419.

22 Statista. "video games: Worldwide statistics.". URL: https://www.statista.com/outlook/dmo/digital-media/video-games/worldwide.

23 Marcus Toftedahl and Henrik Engström. A taxonomy of game engines and the tools that drive the industry. In *DiGRA 2019*, pages 6–10, 2019.

24 Wikipedia. "netcode". URL: https://en.wikipedia.org/wiki/Netcode.

25 Amir Yahyavi, Kévin Huguenin, Julien Gascon-Samson, Jörg Kienzle, and Bettina Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *IEEE ICDCS*, pages 134–144, 2013. doi:10.1109/ICDCS.2013.62.

26 S.F. Yeung, J.C.S. Lui, Jiangchuan Liu, and J. Yan. Detecting cheaters for multiplayer games: theory, design and implementation[1]. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006.*, pages 1178–1182, Las Vegas, NV, USA, 2006. IEEE. doi:10.1109/CCNC.2006.1593224.

27 Su-Yang Yu, Nils Hammerla, Jeff Yan, and Peter Andras. A statistical aimbot detection method for online FPS games. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Brisbane, Australia, 2012. IEEE. doi:10.1109/IJCNN.2012.6252489.

28 Saman Taghavi Zargar, James Joshi, and David Tipper. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys & Tutorials*, pages 2046–2069, 2013. doi:10.1109/SURV.2013.031413.00127.