Journal of Computer Languages 85 (2025) 101370

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Netskip—A novel architecture for consistent multiplayer videogame state

Francesco Bertolotti, Walter Cazzola *, Luca Favalli, Dario Ostuni, Leonardo Secco

Computer Science Department, Università degli Studi di Milano, Italy

ARTICLE INFO

Keywords:
Distributed systems
Conflict-free replicated data types
Distributed multiplayer video games
Eventual consistency

ABSTRACT

In the past decades, video games have experienced a remarkable surge in popularity, making the gaming industry a multi-billion-dollar business. Simultaneously, game development has expanded beyond AAA publishers, with independent developers achieving impressive success. Today's users are highly aware of quality standards and expect a reliable experience, especially in multiplayer games. However, traditional architectures, peer-to-peer and client–server, present complementary limitations, either from a user experience perspective (e.g., stability, latency, disconnection handling) or from a business perspective (e.g., deployment, maintenance, and scalability costs). In this work, we propose a novel architecture—dubbed *netskip*—that bridges the gap between peer-to-peer and client–server models by combining their strengths. The *netskip* architecture eliminates single points of failure and reduces deployment costs by removing the need for replicated servers, while still supporting disconnections, reconnections, and temporary network partitions. It manages game state in multiplayer settings using conflict-free replicated data types, and we formally prove that the *netskip* architecture ensures eventual consistency among nodes. Finally, we show through an empirical study using real video game traces that the *netskip* architecture delivers performance comparable to peer-to-peer and client–server architectures.

1. Introduction

With the growth of multiplayer gaming communities, the demand for platforms capable of supporting thousands of players at the same time is increasing. E.g., League of Legends has a monthly active player base of 100M people.[1] Such numbers are bound to grow as gaming is becoming an ever more popular entertainment trend and an increasingly profitable industry. Therefore, the design of strategies enabling scalability and reliability of online gaming infrastructures is becoming a research priority in both industry and academia [3–5].

The most broadly accepted gaming architecture to deal with the increasing demand is client–server (c/s) [2,6]. Even if feasible, they require costly server-side hardware dedicated to constantly update and maintain the game state for every active players at any given time. Therefore, c/s architectures may be associated with a number of challenges:

1. the cost of deploying, maintaining, and scaling the infrastructure may be unaffordable, especially for non AAA publishers[2];
2. the costs to up-scaling (when the number of players increases) for the acquisition of replicated servers may encourage the adoption of waiting queues resulting in delays and quality of service disruption [7];
3. a server is a single point of failure from which all players depend [7]; server replication and load balancing are usually adopted for massive multiplayer online games but rarely for the preservation of a single match/session state;
4. no service can be provided during software updates.

A common alternative to c/s is the peer-to-peer (p2p) architecture [2,8]. Compared to c/s, p2p architectures are cheaper, scalable and available [3]. Despite these advantages, they still have some issues. A careful game state management is required to ensure state consistency[3] among players. Maintaining this consistency may require the introduction of the deterministic lockstep that has been proven to introduce significant slowdowns [5,7]. Moreover, p2p architectures should be designed to dynamically deal with connection and re-connection of unreliable peers [5,7].

* Corresponding author.
E-mail addresses: francesco.bertolotti@unimi.it (F. Bertolotti), cazzola@di.unimi.it (W. Cazzola), favalli@di.unimi.it (L. Favalli), dario.ostuni@unimi.it (D. Ostuni), leonardo.secco@studenti.unimi.it (L. Secco).

[1] https://www.polygon.com/2016/9/13/12891656/the-past-present-and-future-of-league-of-legends-studio-riot-games

[2] AAA productions are usually funded by major publishers with marketing budgets of over 10 million dollars [1].

[3] The game state is consistent across the network when all replicas are the same. True consistency is hard to maintain. Hence eventual consistency is a more reasonable goal for most use cases [2].

https://doi.org/10.1016/j.cola.2025.101370
Received 7 March 2025; Received in revised form 22 September 2025; Accepted 15 October 2025
Available online 24 October 2025

Usually, both p2p and c/s architectures rely on a matchmaking server to create matches; in the p2p scenario all the players are connected to each other in a full clique whereas in c/s a connection is established with a server hosting the match. Depending on the game genre, a match can be composed of any number of players ranging from 2 to 100. The only exception is represented by massively multiplayer online (MMO) games that must support thousands of players sharing the same game world but this is a quite a niche case as confirmed by looking at the Steam[4] digital store. At the time of writing, the Steam digital store counts 54,579 games with a peak user base of 23M daily players; the 100 most played games account for 20.36% of the total player base. Only 14 games out of the 100 most played games are MMOs and their peak player base amounts to the 2.23% of the total [9]. The five games with the highest player count are Counter-Strike: Global Offensive,[5] Dota 2,[6] PLAYERUNKNOWN'S BATTLEGROUNDS,[7] Apex Legends[8] and Grand Theft Auto V,[9] supporting 10, 10, 100, 60 and 30 players in the same match respectively. None of which can be considered an MMO game.

We argue that alternatives do exist. In this paper, we present a novel architecture—called *netskip*—based on *conflict-free replicated data types* (CRDT) [10] and skip lists [11]. CRDTs are popular data structures aimed to maintain state replicas across distributed nodes. Updates are asynchronous and replicas are proven to eventually converge to a common state. Such properties ensure performance and scalability in distributed systems [10]. CRDTs have already been proven successful for applications such as collaborative editing [12–14]. At the same time, choosing skip lists allows access to any range of elements in expected logarithmic time: the algorithms for insertion and deletion in skip lists are simpler and significantly faster than equivalent algorithms for balanced trees [11]. As a result, the netskip architecture combines CRDTs and skip lists to enable the deployment and management of networks with higher resilience—i.e, the ability to maintain an acceptable quality of service in response to faults—than p2p and better cost-efficiency than c/s while reducing overheads related to consistency checking to a minimum.

Our work answers the following research questions (RQ):

RQ_1. Can the netskip architecture overcome the mentioned issues of c/s and p2p architectures for online gaming?

RQ_2. How do netskip, c/s and p2p architectures compare in terms of performance for online gaming?

To answer these questions we evaluate and compare the netskip architecture performance against c/s and p2p architectures for different network workloads. Network conditions are obtained by setting parameters such as inter-arrival time and packet length based on real-world data.[10] We compare and assess the effectiveness of netskip architecture wrt. the current standards and their respective issues. We pinpoint the trade-offs, if any, that netskip architecture introduces to address the problems related to current online gaming architectures.

Our work does not focus on MMOs due to the lack of publicly available traces and the amount of required computational resources. Nonetheless MMOs are just a niche case that affects a limited share of the total number of players as we mentioned earlier. MMOs network architectures usually rely on load balancing techniques [16–18] that could theoretically be replicated using netskip architecture.

The rest of this paper is organized as follows. Section 2 presents the terminology. Section 3 introduces netskip architecture, its implementation and intended usage. Section 4 describes experiments and results. Lastly, Sections 5 and 6 overview the related works and draw some conclusions respectively.

2. Background

In this section, we overview the main concepts of the netskip architecture. These include state-based CRDTs and skip lists. State-based CRDTs and operation-based CRDTs are theoretically equivalent [10]. Therefore, while the netskip architecture leverages deltas, aligning with operation-based CRDTs, we will refer to state-based CRDTs as they lead to a simpler proof (refer to Section 3.2).

2.1. Conflict-free replicated data type

CRDT [10] is a failure-resilient data structure which keeps several consistent replicas without need of synchronization in a distributed environment. CRDTs have been applied successfully in application domains, such as, collaborative editing and databases. A few examples are the Nimbus Note collaborative editor,[11] SoundCloud Roshi,[12] and Riak NoSQL database.[13] This work focuses on state-based CRDTs [10].

Let $\Pi = \{p_0, \dots, p_{n-1}\}$ be a collection of n interconnected processes part of an asynchronously and unreliably updated network without global-clock or shared memory. Each p_i holds a replicated object (CvRDT), i.e., an object aiming to be consistent with the one held by p_j $\forall j \in \{0, 1, \dots n-1\} \wedge j \neq i$.

Formally, a CvRDT is defined as a *monotonic semi-lattice object*, i.e., a 6-tuple $(S, \leq, s_0, u, q, m)$ [10], where:

1. S is a set of states;
2. $\leq$ defines the partial order on S;
3. $s_0 \in S$ is the initial state for all $p \in \Pi$.
4. u is a method that updates the local object. Additionally $s \leq s \circ u \;\; \forall s \in S$.
5. q is a side-effect free query method used by a process to read the current object state $s \in S$.
6. m is a method that merges the local state s_i with the state from a remote replica s_j by computing their *least upper bound* $m(s_i, s_j)$.

Each process periodically sends its state to all the other processes which merge the state by applying m. Given eventual delivery[14] and termination properties, any replicated monotonic semi-lattice object is proven to be a CRDT [10].

2.2. Skip lists

Skip lists are ordered, probabilistic data structures [11] that enable search, insertion and deletion operations in $O(\log n)$ on average where n is the current number of inserted elements. This is made possible by a layered structure in which each layer is half as likely to be generated as the preceding one: as a result, the depth of the structure tends to be asymptotically logarithmic in the number of elements. On insertion, the new element is first added to the bottom layer and then, with probability .5, to each subsequent layer. The data structure can be navigated either from top to bottom or from left to right through bi-dimensional references among layers and among different elements of the same layer. Skip lists have the same properties of b-trees [11]

[4] https://store.steampowered.com
[5] https://blog.counter-strike.net/
[6] https://www.dota2.com
[7] https://www.pubg.com/
[8] https://www.ea.com/games/apex-legends
[9] https://www.rockstargames.com/V/
[10] All network traces were taken from http://perform.wpi.edu/downloads/#aom as in [15]

[11] https://github.com/yjs/yjs
[12] https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events
[13] https://riak.com/products/riak-kv/riak-distributed-data-types/
[14] Eventual delivery is a property for which an update delivered at some replica is *eventually* delivered to all replicas [10].

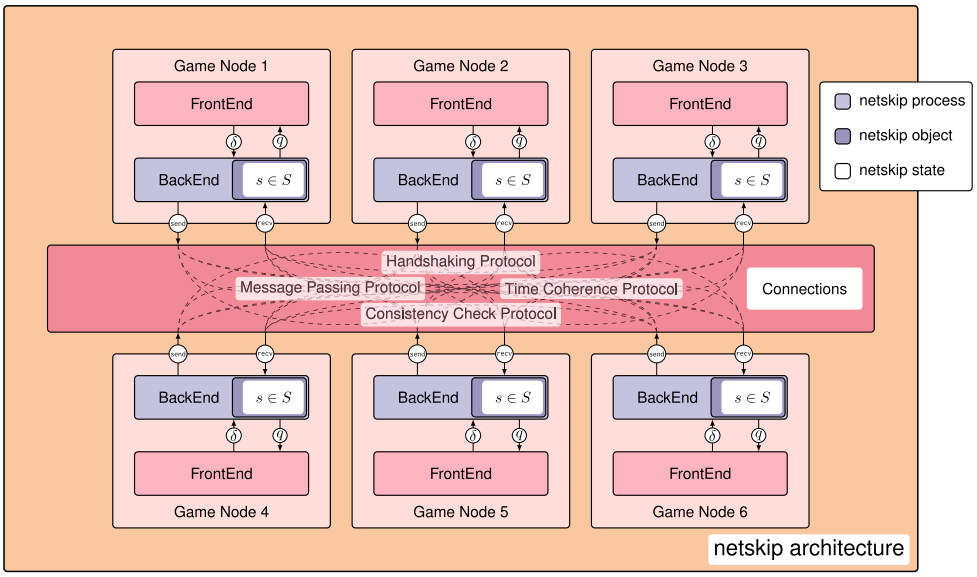


Fig. 1. A snapshot of a game played using a netskip architecture. Six netskip processes run by six players are shown along the various communication protocols.

and can be used in the same applications but they provide additional optimization options for domain-specific use-cases, such as compression and logarithmic range access (see Section 3.3 for implementation details on skip lists).

3. Netskip

In this section, we discuss the netskip architecture in detail. In Section 3.1, we present an overview of all the main aspects of the netskip architecture. Section 3.2 formally defines netskip objects and proves their equivalence to CRDTs. Section 3.3 discusses netskip components and protocols.

3.1. Overview

From now on we will talk of a network as a collection of netskip processes where a netskip process represents a running netskip instance independent of the others. These processes are distributed on several computers and connected to form some kind of topology. The only requirement is that each netskip process is reachable from all other netskip processes (in one or multiple hops) and that messages sent to a process will be eventually delivered.

A netskip network is a specialized case of p2p network that focuses on the management of a game state. Netskip is designed to be used in online games where players interact in real time.

A netskip network is initialized by a single netskip process. Additional processes join by connecting to any existing member of the network. When a new process joins, the *handshaking protocol* is triggered: the initial game state and all accumulated changes (called δs) are transmitted to the joining process. After a successful handshake, the two processes become neighbors.

Fig. 1 shows a high-level view of a distributed game network with six nodes. Each node runs its own game (FrontEnd) instance and represents a player partaking the match. The FrontEnd does not maintain the game state itself but delegates this aspect to the netskip architecture (BackEnd). In each game-frame, the FrontEnd will display the game state to the player according to the game state stored in its netskip process (queried by method `q`). Next, the player will probably perform an action changing the state: a δ will represent such a change. E.g., the generated δ=`{HP:-10,from:2,to:6}` may represent a loss of Health Points (HP) caused from game node 2 to game node 6. Upon generation, this new δ will be:

- stored locally (the δ-labeled arrow in Fig. 1);
- propagated to the neighboring netskip processes.

Upon receiving the δ, a netskip process will:

- do nothing if δ was already stored;
- otherwise, store the δ and propagate it to all its neighbors.

To ensure eventual consistency across all connected nodes, the Netskip architecture employs a flooding algorithm. Each new δ is propagated to neighboring nodes, while outdated δs are discarded. This approach allows nodes to operate with only local knowledge of their immediate neighbors, eliminating the need for global network topology awareness.

Given eventual delivery, this mechanism ensures that a newly generated δ will be propagated to all connected netskip processes. This is only the first piece of the puzzle: δs alone cannot possibly make up a game state in its entirety. You need a state to start updating from. The `forward` function (see Table 1) takes two arguments: a game state and a δ. It applies the given δ to the given game state returning an updated game state. Folding the `forward` function through all the stored δs starting from a common initial game state generates the current game state. As long as two netskip processes have the same δs and the same `forward` function, they will reach the same game state. The δs are folded using the forward function according to an ordering that respects their generation times. Since, δs may originate from different machines, this ordering is enforced by the *time coherence protocol*—based on the Berkeley algorithm [19]—which manages a shared clock across netskip processes. A separate *consistency check protocol* periodically verifies and resolves state inconsistencies between neighboring netskip processes.

The `compress` function reduces the game state size by merging two δs into one. E.g., given δ_1=`{HP:-10, from:2, to:6}` and δ_2=`{HP:-10, from:2, to:6}`, the resulting compressed δ_c will be `compress`(δ_1, δ_2)=`{HP:-20,from:2,to:6}`. We avoid folding through every δ each time the game state is queried by using skip lists to store the representation of a range of δs in a single compressed δ. The `compress` function is used by the *history rewrite mechanism* to limit the growth of δs when the game session becomes too long. The history rewrite mechanism overwrites the initial δs with a compressed δ. A single compressed δ is stored in place of many uncompressed ones saving some space. Note that, while skip lists hold *also* a compressed version of range of δs, history rewrite replace the same range. By enabling history rewrite the netskip architecture loses its property of eventual consistency. While history rewrite can be a useful asset to

optimize performance in long running sessions, it is not required in most cases. For this reason, we assume that history rewrite is disabled in our experimentation and in the following proof.

Further details about the framework, its protocols and its components can be found in Section 3.3.

3.2. Proof that netskip is a CRDT

Let us prove that a netskip object is a CRDT and therefore its replicated objects have the eventual consistency property.

A *netskip object* is the description of the state of a netskip process and the operations it can perform. It is defined as

$$N = (G, \Delta, g_0, f, h, P),$$

where:

1. G is the set of game states from the `FrontEnd` standpoint;
2. Δ is a non-empty set of all possible game state δs—i.e., the transition from one game state to another; Δ is totally ordered wrt. the arbitrary binary relation $\leq$.
3. $g_0 \in G$ is the initial game state;
4. $f : G \times \Delta \to G$ is the function that given the current game state and a δ updates the game state;
5. $h : \Delta \times \Delta \to \Delta$ is the function that computes the composition of two δs (i.e., if $a, b \in \Delta, a \leq b$ then $\forall g \in G\ f(f(g,a),b) = f(g,h(a,b))$);
6. $P \subset \Delta$ is a set of δs representing the current game state from the netskip architecture standpoint.

Then, we define some functions that a netskip process can use to operate on netskip objects:

1. $u : \mathcal{P}(\Delta) \times \Delta \to \mathcal{P}(\Delta)$ is the function that adds a δ into a netskip object;

   $$u(P, \delta) = P \cup \{\delta\}$$

2. $q : \mathcal{P}(\Delta) \to G$ is the function that queries the netskip process `BackEnd` from the `FrontEnd` perspective;

   $$q(P) = \begin{cases} g_0 & \text{if } P = \varnothing \\ f\left(q(P \setminus \{\max(P)\}), \max(P)\right) & \text{otherwise.} \end{cases}$$

   Note that $\max(P)$ is always defined because P is totally ordered;
3. $m : \mathcal{P}(\Delta) \times \mathcal{P}(\Delta) \to \mathcal{P}(\Delta)$ is the functions that merges two netskip objects game states.

   $$m(P, P') = P \cup P'.$$

   The application of the merge function requires two netskip objects N and N' s.t. $N = (G, \Delta, g_0, f, h, P)$ and $N' = (G, \Delta, g_0, f, h, P')$.

Now, the netskip object defined above can be also defined as a 6-tuple $(S, \subseteq, s_0, q, u, m)$ where:

1. $S = \mathcal{P}(\Delta)$;
2. $\subseteq$ is a binary operator that operates on elements of S defined as $A \subseteq B \Leftrightarrow a \in B\ \forall a \in A$;
3. $s_0 = \varnothing$;
4. q, u, m are defined as above.

Let us show that a netskip object is a *monotonic semi-lattice object* [10] by proving properties 1–6 presented in Section 2.1.

(i) $\subseteq$ defines a partial order on $S = \mathcal{P}(\Delta)$ by definition of the subset relation.

(ii) given two states $s, s' \in S$

$$m(s, s') = s \cup s' = \sup(\{s, s'\}).$$

where $\sup(\{s, s'\})$ corresponds to the smallest set containing both s and s'; it is evident that m and sup compute the same function;

(iii) given $s \in S$ and $\delta \in \Delta$ the update function is monotonically non-decreasing by definition:

$$s \subseteq s \cup \delta = u(s, \delta).$$

Shapiro et al. [10] proved that a *monotonic semi-lattice object* is a CRDT, this implies that a netskip object is a CRDT. This theoretical proof ensures that a distributed application using the netskip architecture benefits from the CRDTs property of eventual consistency.

3.3. Implementation

Netskip is a C library implemented in four modules:

- `list` is a skip lists implementation;
- `util` are utilities used by other modules;
- `test_util` utilities to test the netskip architecture and its applications;
- `netskip` is the core of the netskip architecture implementation.

The library amounts to 2857 lines of code (LoC) and 3384 testing LoC, for a total of 1879 executable LoC computed by `meson`. Headers are 1788 LoC.

A netskip process is independent from the application that uses it. That is, the descriptions of in-game objects are arbitrary and left to game developers which should define a mapping between the in-game state description—in the `FrontEnd` application—and the netskip object describing the netskip process. A netskip process is unaware of how the game is implemented rather it focuses on the orchestration of all the components to maintain consistency without loss of generality. Table 1 reports all the elements—parameters and functions—that the game developers must implement. Among those, we discuss δs, functions f and h (`compress`), s_0 (initial game state, and network-specific options as the `send` function.

Time Coherence. Maintaining a coherent clock is mandatory to preserve the correct δs ordering. Each netskip process periodically sends its local clock to its neighbors using ping messages. A distributed version of the Berkeley algorithm [19] updates the local clock using the value of the local clock received by the neighbors. A netskip process leverages the neighbors skip list to store clock information brought by pings and retrieves them in constant time. This protocol is also initiated after the completion of any handshake.

Delta (Δ). Netskip piggybacks each user-generated $\delta \in \Delta$ with the triple (panic epoch, timestamp, id), where id is a random integer identifying the netskip process, timestamp is the time at which δ was passed to the netskip process by the `FrontEnd`, and epoch denotes the number of panics. This information defines a total ordering over Δ. The current game state is obtained by applying all δs in the netskip object to the initial state s_0, which can be done using the forward function f. E.g., a δ could represent Player 1's health point loss (δ={$\mathtt{HP}_1$:-10}), which will cause an update of Player 1's HP ($f(\{\mathtt{HP}_1 : 100\}, \delta) = \{\mathtt{HP}_1 : 90\}$) in the game's local `FrontEnd`. Due to eventual delivery, the same δ will eventually be received by all netskip processes in the network, stored in the respective netskip object and in term update Player 1's HP for all players. The implementation leverages on skip lists to optimize this operation by extracting in constant time a compressed δ equivalent to all the δs in the netskip object. This permits to apply the forward function only once for each query (q). δs are also the only user-generated data sent across the network after the initial handshake protocol. The netskip architecture applies the `after_receive` and `before_send` functions. The game developer can customize `after_receive` and `before_send` to achieve arbitrary goals such as compression, anti-cheat and transmission error. Nonetheless, `after_receive` and `before_send` could also do nothing without affecting netskip's execution.

Skip list. Netskip objects store the δs in a variant of the traditional skip lists (Fig. 2). In our implementation, all layers but the lowest one do not store δs directly but the compressed δs. This allows for a fast

Table 1
Overview of the customization parameters and netskip functions. Running a netskip process requires all entries to be provided. Each function also accepts an additional `user_arg` which can be any arbitrary information provided by the user—e.g., a parameter or a callback—for further customization purpose.

Functions	Description
`forward(g, δ, user_arg)`	*Function that applies the δ to the game state g. It returns the updated game state.*
`send(t, s, m, user_arg)`	*Function to send a new message m of size s to target netskip process t*
`compress(d1st, d2nd, user_arg)`	*Function that computes the compression of δs d_{1st} and d_{2nd}*
`destroy(d, user_arg)`	*Function to destroy a δ data structure d*
`sizer(d, user_arg)`	*Function that returns the size of a δ d*
`after_receive(d, user_arg)`	*Function called upon receiving a δ d*
`before_send(d, user_arg)`	*Function called before sending a δ d*
`user_destroy(g, user_arg)`	*Function that handles the destruction of a game state g*
`delta_hash(d, user_arg)`	*Function that computes a fixed-size hash for a δ d*
`delta_hash_destroy(h, user_arg)`	*Function that destroys an hash structure h*
`delta_hash_compare(h1, h2, user_arg)`	*Function that determines if two hashes h_1 and h_2 coincide*
Parameters	**Description**
`is_new_gamestate`	*True when the current node will start the netskip network*
`initial_game_state`	*Any data representing s_0*
`delta_hash_size`	*The fixed hash size*
`initialize_random_seed`	*True if a random seed should be used*
`seed`	*User-provided seed*
`ping_time`	*Delay between two consecutive pings*
`check_time`	*Delay between two consecutive consistency checks*

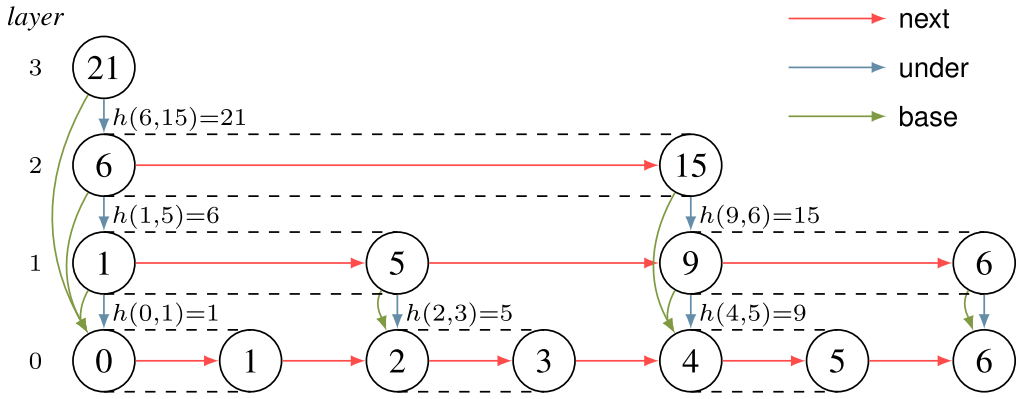


Fig. 2. Compression.

computation of q because ranges of δs can be read from the higher layers of the skip list, which are the first to be accessed when navigating a skip list. Our implementation leverages the multi-layered structure of the skip list data type to keep information regarding compressed δs in higher layers while still having access to each δ. More specifically, *layer 0* (the lowest one) stores all the uncompressed δs that have already been acquired by the netskip object. Following common skip lists implementations, each *layer n* points to *layer 0* (*base*), *layer n-1* (*under*) and to the next element of *layer n* (*next*)—e.g., in Fig. 2, given the skip list element in position 0 of layer 2 (2,0), its *base* is (0,0), *under* is (1,0) and *next* is (2,1). Each pair of consecutive elements on the same layer—such as (2,0) and its *next* (2,1)—identifies a range of elements on the underneath layer. This pair permits to populate an entry of *layer n* with a compressed δ, combining all the δs of the range from *layer n-1*. E.g., element (2,0) in Fig. 2 is the result of the compression between δs from elements (1,0) and (1,1). Moreover, to enable the logarithmic computation of arbitrary δ sub-ranges during consistency checking protocol elements in *layer 0* also refer to the upper layers.

Receive Function and Messages. The receive function is designed as a shared netskip callback to ensure consistent message handling. All netskip messages have the same header. This consists of a magic number, the netskip version and the message type. Upon receiving a message, the magic number and the version are used to check its validity by ensuring they coincide with those stored in the local netskip object. The handling of the rest of the message depends on its type. Netskip uses 18 different type of messages to cover all the protocols, including: handshaking, delta forwarding, time coherence and the checking protocol (see Table 2).

Handshaking Protocol. This protocol governs the connection between two netskip processes. It is designed to ensure that, after the connection is established, each netskip process in the network (i) can perform local handling of adjacent netskip processes' ids—i.e., a netskip process only has local knowledge of the network topology and the netskip process ids are local identifiers instead of being shared across the network—and (ii) is up to date with the rest of the network—i.e., it will eventually contain all δs from both sides of the connection.

Let Π be the set of all netskip processes. Let $N \in \Pi$ be the netskip process requesting a new connection and $M \in \Pi$ the netskip process receiving the request. Let

$$\eta : \Pi \to \mathbb{N} \cup \{\bot\}$$

be the function that maps netskip processes to the network identifiers of their netskip processes. Netskip processes are mapped to $\bot$ when they are not connected to any other process before the execution of the handshaking protocol.

1. $\eta(N) = v_0$ and $\eta(M) = \bot$. N initiates a request for M to join v_0. N adds M to its neighbors, and so does M with N. Then, N welcomes M by sharing s_0 and its δs (as shown in Fig. 3).
2. $\eta(N) = \bot$ and $\eta(M) = v_0$. N initiates a request to join v_0. M adds N to its neighbors, and so does N with M. Then, M welcomes N by sharing s_0 and its δs.

Table 2
Description for all message types.

Protocol name	Messages	Description
Handshaking Protocol	1. HELLO REQUEST 2. HELLO REQUEST EMPTY 3. HELLO REPLY 4. HELLO REPLY EMPTY 5. HELLO REPLY COMPLEX 6. HELLO GAME STATE 7. HELLO DELTA NO FORWARD	The handshaking protocol establishes connections between netskip processes. 1-5 establish a new connection. These messages deal with different scenarios in which a new connection can be established (both inside the network, one inside and one outside, both outside). 6 shares the initial game state s_0. 7 shares current δs with the new adjacent netskip process.
Delta Protocol	1. DELTA	Spreads δs throughout the network.
Checking Protocol	1. CHECK PROTOCOL INIT 2. CHECK PROTOCOL INIT NO MORE 3. CHECK PROTOCOL REPLY 4. CHECK PROTOCOL STANDARD 5. CHECK PROTOCOL SEMISTANDARD 6. CHECK PROTOCOL FINAL 7. CHECK PROTOCOL END 8. PANIC	The checking protocol verifies consistency between netskip processes. 1-3 establish a common search range. 4-6 search for the inconsistency. If any is present, it is shared. 7 concludes the protocol. If an inconsistency is found inside a history rewrite, 8 starts the panic protocol.
Time Coherence	1. PING 2. PING REPLY	Time coherence protocol maintains a coherent clock between netskip processes. 1-2 share their local clock.

3. $\eta(N) = v_0$ and $\eta(M) = v_1$ with $v_0 = v_1$. N initiates a request to create an additional connection with M. E.g., this can be done to achieve a specific topology for v_0. M adds N to its neighbors, and so does N with M.
4. $\eta(N) = v_0$ and $\eta(M) = v_1$ with $v_0 \neq v_1$. In this case, the protocol ends without changes as the merging of different networks is not allowed.
5. $\eta(N) = \perp$ and $\eta(M) = \perp$. In this case, the protocol ends without changes as there is no network to join.

History Rewrite. History rewrite is an optional optimization we introduced to improve space management in the netskip architecture. Please note that this feature breaks the theoretical soundness of CRDTs as old δs will be lost. It is triggered upon exceeding a user given threshold on the number of δs currently stored in the skip list. History rewrite replaces the first two δs in the skip list on *layer 0* with the result of their compression. The same compression is applied to higher layers too if needed—i.e., if the compression of δs on *layer n-1* causes changes in the ranges on *layer n*. The complexity of history rewrite is $O(1)$ in the best case (when the compression involves a single layer) and $O(l)$ in the worst case where l is the number of layers (when the compression affects all l layers).

Checking Protocol. Netskip processes periodically check their neighbors to find inconsistencies in their states. The protocol is designed to exchange only a few messages when there are no inconsistencies. Otherwise, it applies a binary search over a range of δs agreed beforehand to find the inconsistency. The binary search leverages the ability to compute any range of δs in the skip list efficiently thanks to the optimization discussed in the skip list section: skip list layers above 0 store information about a range of δs in the underneath layer. If the inconsistency is caused by an uncompressed δ (i.e., a δ that did not partake in a history rewrite), that δ is sent to the other netskip process. Otherwise, the inconsistency cannot be solved and a panic protocol starts. During the panic protocol, a new common state is randomly selected among the states of all the netskip objects in the network. This state is determined and sent to each netskip process by flooding. Moreover, the panic epoch is increased. Note that the panic protocol is never executed when history rewrite is disabled because inconsistencies can always be solved in this case.

3.4. Limitations

Synchronization. A key component of the Netskip architecture is its clock management system. In the context of video game semantics, maintaining strict action ordering is essential to ensure fairness and consistency in gameplay. Although CRDTs inherently lack ordering guarantees, Netskip addresses this by implementing a distributed variant of the Berkeley algorithm [19] to maintain a shared logical clock. This solution generally provides adequate synchronization; however, it can still result in inconsistencies when faced with network instability.

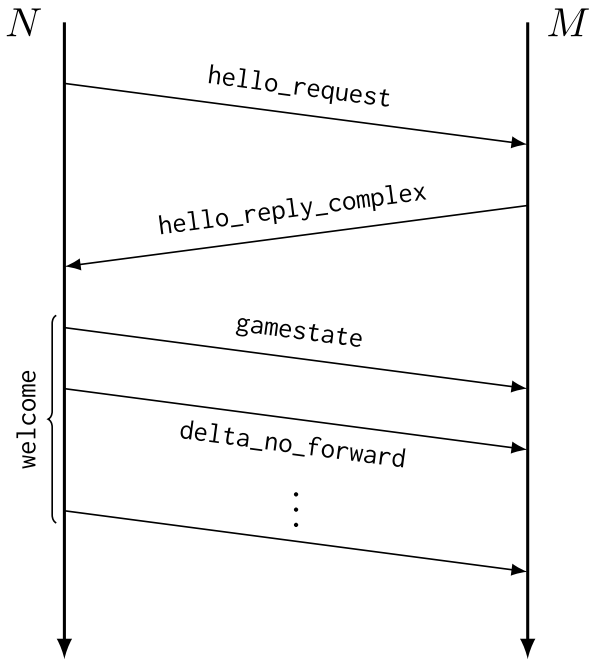


Fig. 3. Handshake protocol on scenario 1.

Scalability. In fact, it is worth noting that the number of deltas sent across the network may saturate the network bandwidth depending on the number of players and type of game.

Usability. Integrating Netskip into existing game architectures may require significant effort, especially if the game was not originally designed with a Netskip architecture in mind. This can lead to increased complexity in the development process, as developers must either implement the game with Netskip from scratch or adapt their code to fit the Netskip architecture.

Debugging. Debugging Netskip-based games can be more challenging than traditional architectures. The distributed nature of Netskip, combined with the use of CRDTs, can make it difficult to trace and resolve

issues that arise during gameplay. Developers may need to invest additional time and effort in creating robust debugging tools and processes to effectively diagnose and fix problems.

Security. Netskip does not inherently provide security features, such as anti-cheat. Developers must implement additional security measures to protect against cheating, hacking, and other malicious activities that could compromise the integrity of the game. This may involve creating custom protocols or integrating existing security solutions on top of the Netskip architecture.

4. Evaluation

To address our research questions, we measure the following metrics:

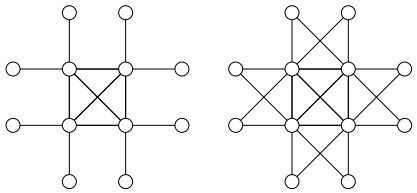
- **Messages distribution by topology**: the number of messages sent by each node in the network, grouped by message type and topology. This metric can be used to assess the overhead introduced by the netskip synchronization protocols over different network topologies.
- **Message distribution by number**: the number of messages sent over time for each topology. This metric can be used to assess the overhead of the flooding algorithm across different topologies.
- **Delta life span**: the time a generated δ lingers in the network before it stops being retransmitted. This metric is an index of the overall bandwidth used by each δ and ensures the number of messages in the network does not diverge over time.
- **Delta receive delay**: the delay between the instant a δ is generated and when it is received by all other users. This metric assesses the perceived lag by the user and is also an estimate for the delay before netskip replicated objects reach consistency.

To compute these metrics, we performed a set of experiments on the netskip architecture and compared it with p2p and c/s architectures, measuring the following quantities:

- the type and number of exchanged messages ($\#m$), and
- the δ delivery time (δ_t) for both sending and receiving.

We simulated workloads associated to popular video game matchmaking assets. These include number of players $(10, 20, 40, 100)$ and average ping delay (online/LAN). Furthermore, we tested additional topologies—meshstar and meshstar2 (Fig. 4)—which are valid netskip topologies but cannot be created when using p2p nor c/s architectures. Each experiment was run for 10 s since the behavior does not significantly change over time.

Although hybrid architectures improve upon the standard ones (p2p and c/s) [20], they usually rely on balancing techniques among peers or servers [2,21] and their usage is limited to MMOs. Scaling to thousands of players is usually supported by the assumption of a specific world structure or the presence of super-peers [17,20,22,23]. For example, one technique that is often employed to support a large number of players is zoning: the game world is chunked into zones, each hosted by one server or a super-peer [24–27]. When a player moves from a zone to another, the player is migrated to the respective server. There are several techniques that either improve or build on top of this concept [28–32]. Instead, with the netskip architecture, we want to propose an underlying mechanism to provide consistency among peers without preventing the application of these techniques to increase scalability. A game world could be a collection of networks based on the netskip architecture, each representing a single zone and players could be migrated from a netskip network to another based on their position in the game world. This has the practical advantage of removing the need of the physical servers without relying on elected super-peers [33–35] in a p2p network. Nonetheless, this work focuses on different aspects of multiplayer games instead of the support for MMOs thus we limit our comparison exclusively to p2p and c/s architectures.



(a) Meshstar (b) Meshstar2

Fig. 4. Some topologies in a network with 12 nodes.

In particular, Section 4.1 presents the general experimental setup and the experimental process. Section 4.2 presents the experiment results. In Section 4.3 and Section 4.4, we discuss the results and report threats to validity respectively.

4.1. Experimental setup

A Docker container to repeat the experiment is available on Zenodo: https://doi.org/10.5281/zenodo.14987118

Hardware setup. We used a machine with a processor i7-7700HQ at 2.8 GHz and 16 GB RAM at 2133 MHz.

Software setup. Emulating all possible game scenarios is unfeasible, as the variety of the game genres and their scale greatly affects the performance. In order to collect the network-related metrics that allow a proper comparison between netskip and other architectures, we developed a custom simulator written in C. The main execution steps performed by the simulator are summarized in Fig. 5. For more implementation details, please refer to open source implementation available at https://doi.org/10.5281/zenodo.14987118. The simulator emulates a network of nodes that exchange messages according to user-defined JavaScript logic. It advances in discrete time steps, delivering scheduled events, generating statistics, and producing compressed JSON logs of the simulation state. It can be used to simulate any network architecture included p2p, c/s and netskip. The simulator enables the execution of multiple nodes at a time by performing memoization and mocking of δs. There is no correlation between simulation time and time in the host machine: the simulator always runs at the highest possible speed enabling the simulation of multiple game seconds per second. The simulator tracks received messages, sent messages, linked and unlinked nodes for each millisecond of execution (tick). Using the same simulator allows for a fair and consistent comparison among all architectures, thus answering our research questions.

Data setup. The simulator was fed with real-world data from [15][15]: every δ size is extracted from an empirical distribution whereas latency is extracted from the normal distribution $\mathcal{N}(\mu, 3)$ with μ equals to the European mean for the Internet network latency [36]. Another set of experiments was run with arbitrary values from $\mathcal{N}(2, 1)$ to simulate a LAN setting. We argue that latency does not represent an impacting factor over the comparison, being shared between all considered architectures. The charts in Figs. 6–9 are associated to the first set of experiments (European mean latency) and the 40 nodes topology variants.[16] We chose the charts that better highlight the distribution of the results in different setups when working with higher latency and variance due to the ticks being discrete. The other scenarios show comparable results. The results are measured in ticks (multiples of milliseconds): working with smaller values—as in the LAN setting—may cause loss of information due to sensitivity. The choice of a millisecond as time unit is arbitrary but reasonable with respect to the

[15] http://perform.wpi.edu/downloads/#aom

[16] The remaining scenarios are not shown in this paper due to space limitations and are available at https://doi.org/10.5281/zenodo.14987118

1. **Initialization**
   - Allocates memory for `nodes` and `netskip` structures.
   - Sets up skiplists for deltas, messages, and arrival times.
   - Initializes Duktape (JavaScript engine) for scripting node behavior.
   - Registers C functions as global JavaScript functions (e.g., `create_node`, `deliver_message`).
2. **Loading Scripts**
   - Loads and evaluates JavaScript files (node logic, event generators, etc.).
3. **Simulation Loop**
   - For each tick (simulation time step):
     - Updates the random seed.
     - Handles bookkeeping and statistics.
     - Dispatches messages scheduled for delivery at this tick.
     - Calls the JavaScript event generator for custom events.
     - Collects node states and events into a JSON object.
     - Compresses and writes JSON output to a file (with GZIP).
4. **Cleanup**
   - Frees all allocated memory and destroys objects.

Fig. 5. Network simulator execution steps for all architectures.

reaction time of human players, which is about 300 ms on average [37]. Table 3 aggregates the results of the remaining scenarios—i.e., different number of nodes and latency.

4.2. Results

We collected and analyzed the following metrics.

Messages distribution by topology. Regardless of the chosen architecture, the multiplayer setup of distributed games requires each state update to be transmitted to all the nodes in the network. The netskip architecture uses only one out of the 18 defined message types in the transmission of game state updates (Table 2). All the remaining messages are needed to support the netskip protocols from Section 3.3. Notice that c/s and p2p architectures only send game state updates instead. Introducing this distinction allows for a precise evaluation of the overhead caused by the consistency checking and the other netskip protocols. Fig. 6 reports the number of sent messages and their frequency for each message type. In all setups well over 90% of transmitted data is associated to δs, followed by ping and ping reply messages. The remaining message types represent less than 0.1% of the total. This result proves that the overhead of maintaining consistency in an arbitrary network topology using the netskip architecture is below 10% if ping is considered and below 1% otherwise.

Message distribution by number. As introduced in Section 3.1, the netskip architecture uses a flooding routing algorithm to share the δs with all the nodes in the network. This approach allows adopting several arbitrary network topologies but it may cause substantial increase of unnecessary repeated δs sent when applied to topologies with a high number of connections per node. This experiment assesses the netskip architecture performance wrt. the number of messages sent over time for several topologies and compares them against c/s and p2p architectures. Results are shown in Fig. 7; the horizontal blue line represents the mean value. The results show that the netskip architecture favors sparse topologies. Additionally, the top-right corner of each graph displays the average downtime—i.e., the number of milliseconds between two consecutive messages.

Delta life span. We assess the time a generated δ lingers in the network before it stops being retransmitted. This metric is an index of the overall bandwidth used by each δ and ensures the number of messages in the network does not diverge over time. Results for this experiment are shown in Fig. 8, which includes all the quartiles: each quartile indicates that a specific δ has traveled across the respective percentage of connections whereas the maximum is its actual life span. With this metric we highlight that the bandwidth usage of the netskip architecture is comparable with c/s; p2p performs better but with a less smooth experience, represented in Fig. 8 by gaps in which no additional user inputs can be processed until the next lockstep.

Delta receive delay. Delta receive delay is the most important metric from a user standpoint. The perceived lag depends on the delay between the instant a δ is generated and when it is received by all other users. For example, a delta receive delay of 100 ms will cause a time gap between a player action and the visual update for other players of at least 100 ms. This result is also an estimate for the delay before netskip replicated objects reach consistency. Fig. 9 reports all the quartiles for the delay of a δ being received by each node after it was generated at a given millisecond of the simulation.

4.3. Discussion

The experimental results in the previous section evaluated the netskip architecture characteristics and compared its applicability with c/s and p2p. Now we dig deeper into the meaning of the results with the intent of addressing RQ_1 and RQ_2.

To answer RQ_1, we explain how the netskip architecture overcomes existing issues in other architectures. Netskip improves over the c/s architectures by removing the single point of failure and having an adaptable topology. The evaluation was performed on fixed topologies but each node starts generating δs right after its creation and later adapts when new nodes are added. This phenomenon is shown by the initial spike in Figs. 8 and 9 for the netskip-clique topology in which the delay is due to some of the netskip processes not having completed the handshaking protocol yet and receiving the δs at a later time. The same delay is not experienced again in any moment of the execution. Netskip can improve on the scalability and maintenance costs when compared to the c/s architectures while providing for reliability needs

Table 3
Experiment results summary for all topologies and metrics.

Topology	Metric	# nodes (LAN)				# nodes (EU)			
		10	20	40	100	10	20	40	100
netskip-clique	% δ	86.382	92.67	96.248	98.512	85.701	92.743	96.35	98.526
	# messages (1/ms)	11.78	86.975	661.152	10 230.863	11.62	87.61	673.955	10 221.132
	life span (ms)	6.865	7.442	8.044	9.147	66.293	68.907	71.207	73.964
	receive delay (ms)	2.083	2.146	2.301	2.811	28.994	29.532	30.271	31.404
netskip-star	% δ	76.712	85.976	92.481	96.86	76.135	85.981	92.47	96.89
	# messages (1/ms)	1.628	5.299	18.924	106.924	1.635	5.33	18.278	105.847
	life span (ms)	5.561	6.089	6.559	7.264	58.314	61.432	63.621	66.35
	receive delay (ms)	4.096	4.275	4.477	5.002	54.006	56.347	58.016	60.363
netskip-meshstar	% δ	76.806	86.594	93.539	97.97	76.734	86.395	93.445	97.763
	# messages (1/ms)	1.629	5.381	24.327	322.662	1.666	5.327	23.761	222.862
	life span (ms)	5.437	7.779	9.064	10.378	59.105	87.695	91.521	96.115
	receive delay (ms)	3.975	5.109	6.013	6.95	54.802	65.548	82.427	87.104
netskip-meshstar2	% δ	83.705	90.393	95.058	98.025	82.074	90.559	95.098	97.986
	# messages (1/ms)	3.845	13.982	57.547	365.757	3.794	13.758	51.795	336.333
	life span (ms)	7.364	8.266	9.994	11.802	84.109	88.7	92.677	123.827
	receive delay (ms)	3.21	3.644	4.568	5.991	49.925	53.307	56.272	71.089
peer-to-peer	# messages (1/ms)	0.332	1.051	3.572	19.897	0.3	1.092	3.97	21.61
	life span (ms)	3.545	3.888	4.189	4.504	32.329	33.62	34.461	35.537
	receive delay (ms)	2.054	2.056	2.007	2.0	27.955	28.016	27.999	28.021
client–server	# messages (1/ms)	1.006	3.908	16.066	100.2	1.004	3.958	15.643	100.253
	life span (ms)	5.691	6.176	6.564	7.261	61.097	62.478	64.243	66.576
	receive delay (ms)	4.284	4.36	4.482	4.986	56.861	57.364	58.553	60.544

since it does not require powerful server machines to be ran and its consistency is theoretically sound. Netskip can operate over a clique or a meshstar2 topology: both eliminate the single point of failure. Fig. 7 highlights that using the netskip architecture over a clique topology the number of sent message scales quadratically in the number of nodes: in our experiment, a network with 40 nodes generates about 0.4 δs per millisecond; $0.4 \cdot 40^2 = 640$ which is approximately equivalent to the mean value reported in Fig. 7 and Table 3. Moreover, the number of sent messages over meshstar or a meshstar2 topology scales quadratically in the number of nodes in the inner clique. The number of sent messages can be easily kept linear as long as the clique size is $O(\sqrt{n})$ where n is the total number of nodes in the network. We argue that the meshstar2 topology is the best alternative between the two since it removes the single point of failure.

Compared to p2p, netskip leverages CRDTs properties to ensure game state consistency without the need of deterministic lockstep, thus removing the restriction of the game state progressing at the speed of the slowest peer. The effect of the deterministic lockstep on game progression is highlighted by the average downtime in Fig. 7: consistency in netskip topologies does not affect game progression, in fact the average downtime is comparable to c/s. Conversely in p2p the game can progress only when the consensus is reached by all the peers. This causes a downtime about 10 times higher then netskip and c/s which effects are clearly visible by the gaps between consecutive messages in Figs. 7, 8 and 9.

The netskip checking protocol ensures the game state consistency is provided regardless of the topology. Despite having a single point of failure, even the star and meshstar topologies can handle disconnections and topology changes as long as a connected graph is re-established as soon as possible.

RQ_2 is answered by analyzing the overheads the netskip architecture introduces to overcome the limits of c/s and p2p architectures. Fig. 6 highlights that, regardless the chosen topology, less than 10% of transmitted messages are associated to netskip-specific protocols, most of which are ping-related messages, since all the remaining message types combined amount to less than 1% of the total bandwidth. This means that the overhead associated to protocols is limited: maintaining consistency in the netskip architecture costs a 10% increase in bandwidth usage if ping messages are considered and 1% otherwise. The overhead is further reduced in larger networks and long-running simulations (see Table 3). Regarding the number of sent messages, displayed in Fig. 7, the netskip architecture is comparable with c/s in all topologies except for clique and provides the same performance when choosing the star topology. In the p2p architecture the number of sent messages is considerably lower because the peers must wait for the deterministic lockstep before being able to send a new δ, thus the game is subject to a slower progression as previously discussed. Delta life span is a metric for network workload. Note that the most important factor is the maximum value in the graph. The results shown in Fig. 8 highly depend on the topology. In fact c/s and star yield similar results whereas meshstar and meshstar2 introduce overhead due to an increased number of hops for each δ on average. The same applies to clique and p2p: delta life span in clique is about doubled compared to p2p due to the flooding algorithm but the game progression is smoother. Fig. 9 highlights lag perception the user experiences during a game. Again, this metric depends on the network topology, thus the best performance is achieved by p2p and clique, which minimize the number of hops. Star, meshstar2 and c/s topologies also have comparable results with each other and outperform meshstar.

Given the acquired data, the results reported in Table 3 and the performed analysis, meshstar2 shows the best performance among the simulated topologies for the netskip architecture. Moreover, it overcomes the issues in existing architectures while reducing overheads to a minimum. Netskip can overcome existing issues in c/s and p2p by supporting scalable, reliable and cost-efficient architectures for distributed games. Netskip performance is comparable with other architectures but some topologies may aggravate some of the bottlenecks. The best of both worlds is met when using the netskip architecture over unconventional topologies such as meshstar2, but the best topology should be considered on a case by case scenario.

4.4. Threats to validity

In this section, we discuss threats that may have affected our evaluation or our conclusion.

Internal Validity. Both c/s and p2p are general communication architectures. Therefore, a different implementation may yield different results. We used the same netskip framework to instantiate netskip networks as well as c/s and p2p networks (as discussed in Section 4.1). This approach has the advantage that all architectures rely on the same

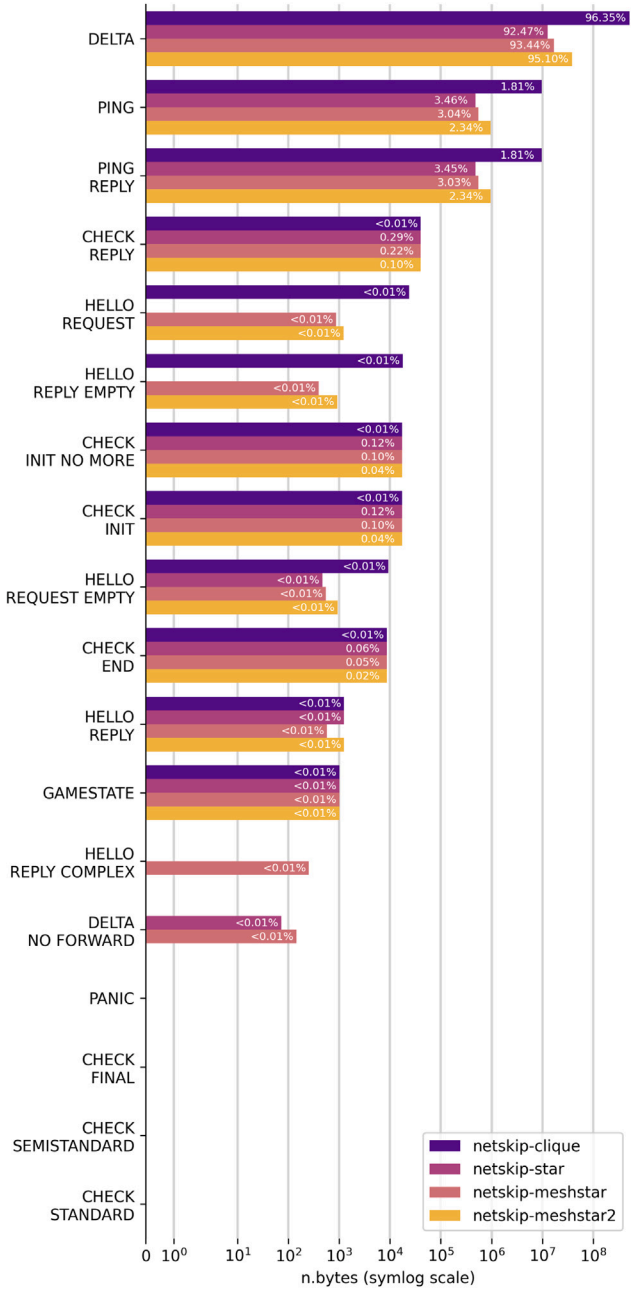


Fig. 6. Number of sent bytes grouped by message type and topology (lower is better). Each color represents a topology; messages across all types sum up to 100%. The higher the percentage of non-delta messages, the higher the overhead of using netskip.

implementation. At the same time, we cannot exclude the possibility to have introduced unintended biases. To mitigate this issue, the simulator was configured to disable any netskip specific protocol when running in c/s or p2p mode—although real implementation for c/s and p2p probably have their own protocols. Ultimately, we are confident to have conducted a fair comparison between all the architectures.

External Validity. It is also important to note that different data traces may yield to different results. To avoid any bias in favor of netskip, we included results from third-party public data on which we have no control. We also used real data to avoid any bias due to *in vitro* data preparation. We did not include hybrid architectures in our evaluation. This may lead to different results, however hybrid architectures are usually tailored for specific applications and focus on the scalability of MMO games which netskip is not trying to address.

5. Related work

In this section, we discuss some relevant works in the field of CRDTs and in the field of distributed multiplayer games that can be in some way related to our work.

CRDTs. Research on CRDTs is very active. They can be used for the development of collaborative editing applications [12,38–40]. In the same application domain, Castiñeira and Bieniusa [41] address temporary connection unavailability. A solution to the gossip problem using CRDTs was proposed in [42]. Mukund et al. [43] proposed an implementation for OR-sets using CRDTs. A mechanism for data causality tracking is proposed in [44]. Kaki et al. [45] proposed a framework for correct-by-construction synthesis of merge functions for replicated data types. Other works focus more on the consistency checking aspect of CRDTs such as the traceability of growing partitionable ADTs [46]. By extension, even more, research focuses on replicated data in general, both for replicated and distributed big data [47] and convergence of concurrent real-time applications [48]. Lastly, Barbosa et al. [49] developed a privacy-preserving protocol for CRDTs.

Replication management. Usually, to maintain game objects consistent, updates are only allowed on primary replicas stored on coordinators. Coordinators are elected among peers and store all primary replicas for a given region in [17,50–52]. However, specific regions may become overloaded with entities. In this case, GauthierDickey et al. [53] propose to dynamically split overloaded regions into smaller ones. Instead of splitting regions, Iimura et al. [54] propose to let coordinators delegate tasks (such as AI calculations) to other players. Frey et al. [55] split entities into Voronoi regions and elects a coordinator among peers. A similar approach is used by the netskip architecture during panic. In [56], to improve security, peers cannot influence their chances of storing a replica. Chang et al. [57] design a replication management system for cloud-based architectures. A mechanism based on the equity theory is proposed by Shen et al. [58].

Non-primary replicas are then distributed to the players usually according to an Area-Of-Interest (AOI). Replicas can be pre-fetched before they appear in the AOI [59]. Carlini et al. [60] address the same problem by using a gossip-based approach.

Consistency control. Many games perform updates on local replicas optimistically and provide consistency control mechanisms [2]. Chandler et al. [61] allow small inconsistencies to have a faster progression. Inconsistencies are resolved using an arbitrator and by "amalgamating" states. A similar level of inconsistencies is tolerated also in [17,62]. Both use an authoritative replica which is continually propagated to all players. A publish–subscribe mechanism is used in [3]. Here, players subscribe to publishers that are in their area-of-interest. Timewarp [63] originally developed for database systems and then adapted for games [64] allows immediate local updates, however, a rollback may happen if an inconsistency is detected. Hydra [8] implements serverless consistency in a p2p architecture using checkpoints and restoration. Xu and Wah [65] delay action by introducing a local lag to solve the reordering problem but assumes the maximum lag between any two players is known. Other techniques accept a little degree of inconsistency to reduce latency by introducing behavioral or statistical models to interpolate updates [66,67]. Similarly to the previous ones, an information-based approach is taken by [68]. These last three techniques could also be applied in combination with the netskip architecture.

The above application management and consistency control approaches deal with problems of distributed gaming applications that are close to those we are trying to face using the netskip architecture. Netskip also uses an approach similar to [61] for the election of a leader during the panic protocol.

To the best of our knowledge, no other framework or architecture use CRDTs for distributed coordination in online gaming.

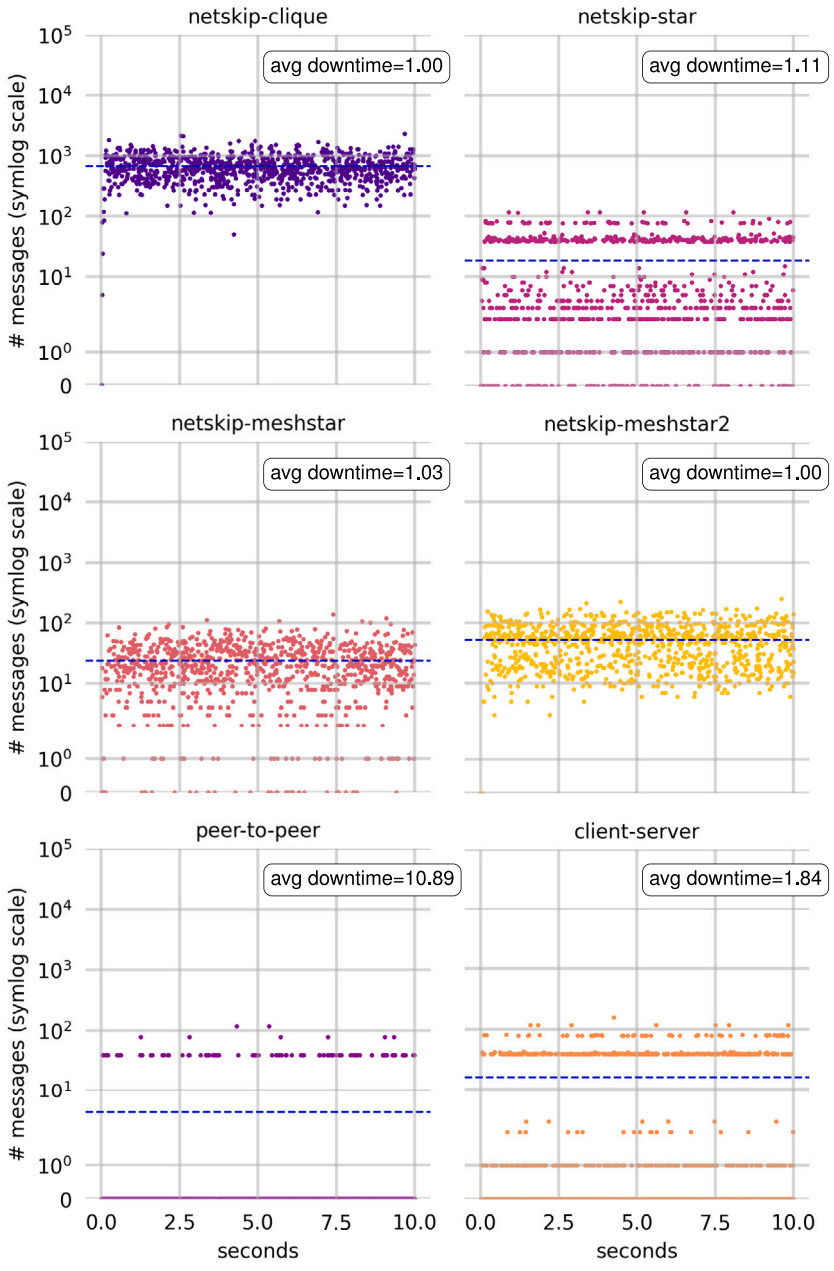


Fig. 7. Messages sent over time by considered topologies (lower is better). The dashed blue line is the mean number of sent messages. The average downtime is the number of milliseconds between two consecutive messages. Note that lockstep is in use for the p2p architecture.

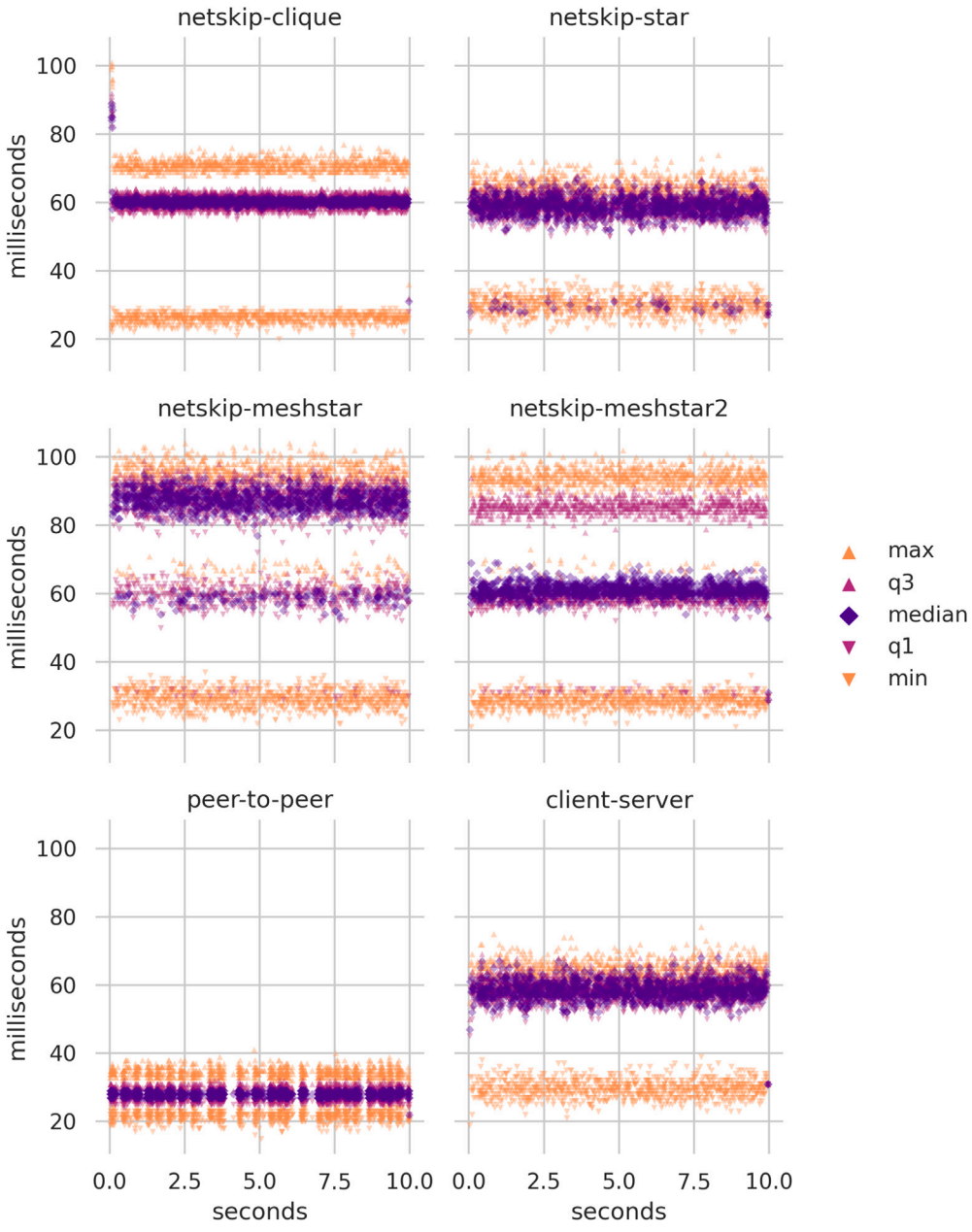


Fig. 8. Delta life span over time for considered topologies (lower is better). Note that lockstep is in use for the p2p architecture.

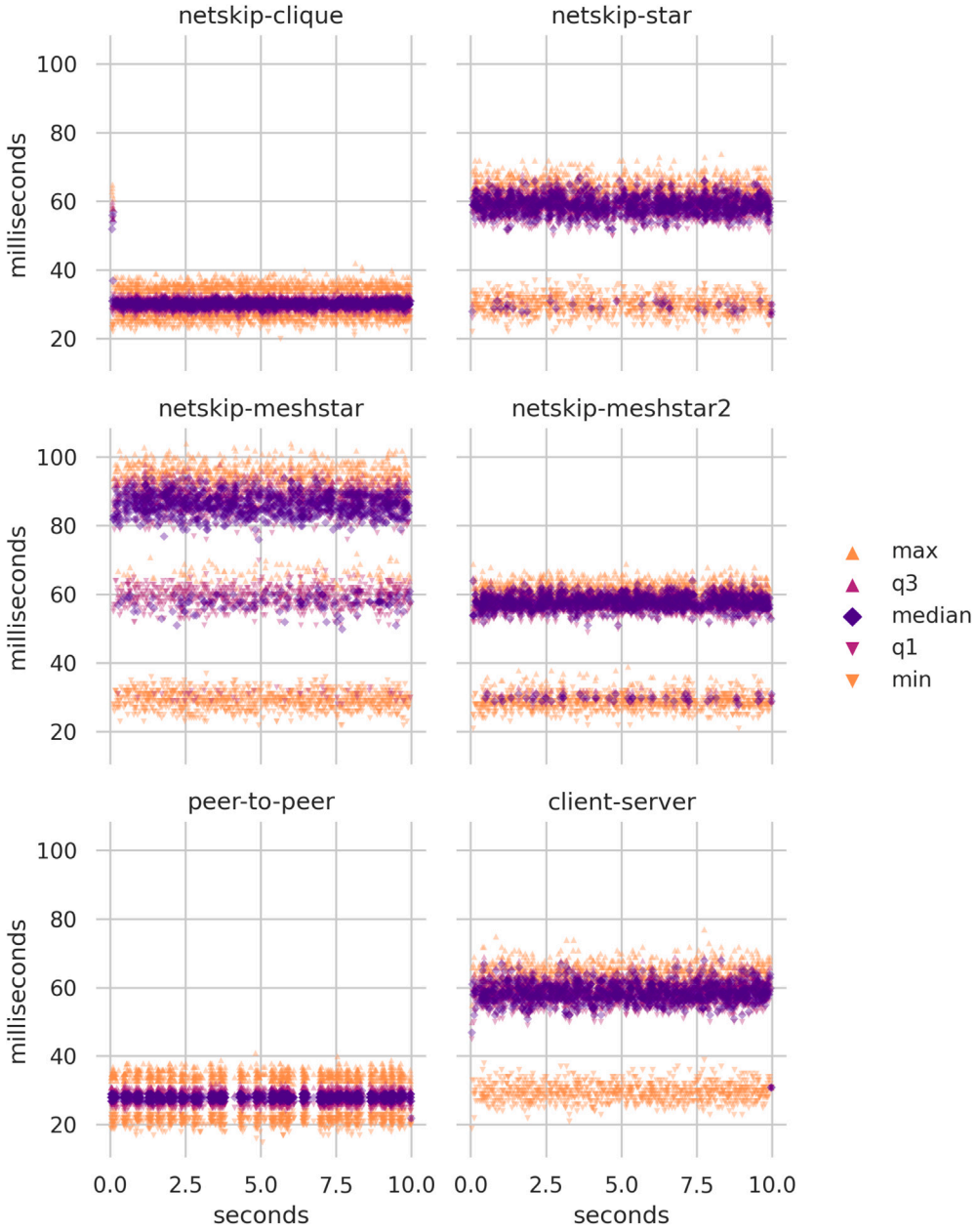


Fig. 9. Delta receive delay over time per topologies (lower is better). Note that lockstep is in use for the p2p architecture.

6. Conclusions

We introduced the *netskip* architecture as a novel network architecture for online gaming. The framework is theoretically grounded and shows potential improvements over existing technologies in terms of reliability and cost efficiency. Our research aimed to address RQ_1 and RQ_2 through an empirical study. The evaluation demonstrates that the trade-offs required to achieve these benefits are modest and likely acceptable in most scenarios, although c/s and p2p architectures still outperform it in specific cases. Netskip offers several customization options, enabling flexible design for application-specific needs. We highlight its comparatively low overhead, especially in terms of message count and bandwidth consumption introduced by netskip-specific protocols. Its performance is particularly strong in non-mainstream topologies. Based on our experimental results, we conclude that the *netskip* architecture is a viable alternative to c/s and p2p models for targeted applications.

CRediT authorship contribution statement

Francesco Bertolotti: Writing – original draft, Software, Methodology, Data curation, Conceptualization. **Walter Cazzola:** Writing – original draft, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Luca Favalli:** Writing – original draft, Software, Methodology, Investigation, Data curation, Conceptualization. **Dario Ostuni:** Software. **Leonardo Secco:** Software.

Declaration of competing interest

We wish to confirm that there are not known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Data availability

A replication package is hosted on Zenodo.

References

[1] S.M. Steinberg, Videogame Marketing and PR, iUniverse, Inc., 2007.
[2] A. Yahyavi, B. Kemme, Peer-to-peer architectures for massively multiplayer online games: A survey, ACM Comput. Surv. 46 (1) (2013).
[3] P. Mildner, T. Triebel, S. Kopf, W. Effelsberg, Scaling online games with NetConnectors: A peer-to-peer overlay for fast-paced massively multiplayer online games, Comput. Entertain. 15 (3) (2017) 3:1–3:21.
[4] Y. Deng, Y. Li, X. Tang, W. Cai, Server allocation for multiplayer cloud gaming, in: B. Huet, A. Kelliher, Y. Kompatsiaris, J. Li (Eds.), Proceedings of the 24th International Conference on Multimedia, MM'16, ACM, Amsterdam, The Netherlands, 2016, pp. 918–927.
[5] B. Bryant, H. Saiedian, An evaluation of videogame network architecture performance and security, Comput. Netw. 192 (2021).
[6] X. Che, B. Ip, Packet-level traffic analysis of online games from the genre characteristics perspective, J. Netw. Comput. Appl. 35 (1) (2012) 240–252.
[7] C. Neumann, N. Prigent, M. Varvello, K. Suh, Challenges in peer-to-peer gaming, ACM SIGCOMM Comput. Commun. Rev. 37 (1) (2007) 79–82.
[8] L. Chan, J. Yong, J. Bai, B. Leong, Hydra: A massively-multiplayer peer-to-peer architecture for the game developer, in: G. Armitage (Ed.), Proceedings of 6th Workshop on Network and System Support for Games, NETGAMES'07, ACM, Melbourne, Australia, 2007, pp. 37–42.
[9] Valve Corporation, Steam & game stats, 2021, available at https://store.steampowered.com/stats/. Last (Accessed 08 June 2021).
[10] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: V.V. Défago Xavier, Petit Franck (Eds.), Proceedings of the Symposium on Self-Stabilizing Systems, SSS'11, in: Lecture Notes in Computer Science 6976, Springer, Grenoble, France, 2011, pp. 386–400.
[11] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, Commun. ACM 33 (6) (1990) 668–676.
[12] S. Weiss, P. Urso, P. Molli, Logoot-Undo: Distributed collaborative editing system on P2P networks, IEEE Trans. Parallel Distrib. Syst. 21 (8) (2010) 1162–1174.
[13] M. Letia, N. Preguiça, M. Shapiro, CRDTs: Consistency Without Concurrency Control, Research Report RR-6956, INRIA, 2009.
[14] B. Nédelec, P. Molli, A. Mostefaoui, E. Desmontils, LSEQ: An adaptive structure for sequences in distributed collaborative editing, in: K. Marriott (Ed.), Proceedings of the Symposium on Document Engineering, DocEng'13, ACM, Florence, Italy, 2013, pp. 37–46.
[15] M. Claypool, The effect of latency on user performance in real-time strategy games, Comput. Netw. 49 (1) (2005) 52–70.
[16] C.E. Benevides Bezerra, C.F. Resin Geyer, A load balancing scheme for massively multiplayer online games, Multimedia Tools Appl. 45 (2009) 263–289.
[17] H.A. Engelbrecht, J.S. Gilmore, Pithos: Distributed storage for massive multi-user virtual environments, ACM Trans. Multimed. Comput. Commun. Appl. 13 (3) (2017) 1–33.
[18] S. Farlow, J.L. Trahan, Periodic load balancing heuristics in massively multiplayer online games, in: S. Dahlskog, S. Deterding, J. Font, M. Khandaker, C.M. Olsson, S. Risi, C. Salge (Eds.), Proceedings of the 13th International Conference on the Foundations of Digital Games, FDG'18, ACM, Malmö, Sweden, 2018, pp. 1–10.
[19] R. Gusella, S. Zatti, The accuracy of the clock synchronization achieved by TEMPO in berkeley UNIX 4.3BSD, IEEE Trans. Softw. Eng. 15 (7) (1989) 847–853.
[20] S.A. Abdulazeez, A. El Rhalibi, D. Al-Jumeily, Dynamic area of interest management for massively multiplayer online games using OPNET, in: H. Hamdan, D. Al-Jumeily, A. Hussain, H. Tawfik, J. Hind (Eds.), Proceedings of the 10th International Conference on Developments in ESystems Engineering, DESE'17, IEEE, Paris, France, 2017, pp. 50–55.
[21] I. Barri, C. Roig, F. Giné, Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability, Multimedia Tools Appl. 75 (2016) 2005–2029.
[22] O. Beaumont, A.-M. Kermarrec, L. Marchal, E. Rivière, VoroNet: A scalable object network based on voronoi tessellations, in: D.K. Panda (Ed.), Proceedings of the 21st International Parallel and Distributed Processing Symposium, IPDPS'07, IEEE, Long Beach, CA, USA, 2007, pp. 1–10.
[23] C. Anthes, P. Heinzlreiter, J. Volkert, An adaptive network architecture for close-coupled collaboration in distributed virtual environments, in: Y. Cai, J. Brown (Eds.), Proceedings of the International Conference on Virtual Reality Continuum and Its Applications in Industry, VRCAI'04, ACM, Singapore, 2004, pp. 382–385.
[24] P. Moll, S. Isak, H. Hellwagner, J. Burke, A quadtree-based synchronization protocol for inter-server game state synchronization, Comput. Netw. 185 (11) (2021).
[25] Y. Deng, R.W.H. Lau, Dynamic load balancing in distributed virtual environments using heat diffusion, ACM Trans. Multimed. Comput. Commun. Appl. 10 (2) (2014) 1–19.
[26] S.-Y. Hu, C. Wu, E. Buyukkaya, C.-C. Chien, T.-H. Lin, M. Abdallah, J.-R. Jiang, K.-T. Chen, A spatial publish subscribe overlay for massively multiuser virtual environments, in: Proceedings of the International Conference on Electronics and Information Engineering, ICEIE'10, IEEE, Kyoto, Japan, 2010, pp. 314–318.
[27] E. Buyukkaya, M. Abdaliah, R. Cavagna, VoroGame: A hybrid P2P architecture for massively multiplayer games, in: J.F. Buford (Ed.), Proceedings of the 6th Consumer Communications and Networking Conference, CCNC'0), IEEE, Las Vegas, NV, USA, 2009, pp. 1–5.
[28] X. Jiang, F. Safaei, Supporting a seamless map in peer-to-peer system for massively multiplayer online role playing games, in: E. Elmallah, M. Younis (Eds.), Proceedings of the 33rd Conference on Conference on Local Computer Networks, LCN'08, IEEE, Montréal, QC, Canada, 2008, pp. 443–450.
[29] S.-Y. Hu, S.-C. Chang, J.-R. Jiang, Voronoi state management for peer-to-peer massively multiplayer online games, in: B. Wei (Ed.), Proceedings of the 5th Consumer Communications and Networking Conference, CCNC'08, IEEE, Las Vegas, NV, USA, 2008, pp. 1134–1138.
[30] S. Shen, S.-Y. Hu, A. Iosup, D. Epema, Area of simulation: Mechanism and architecture for multi-avatar virtual environments, ACM Trans. Multimed. Comput. Commun. Appl. 12 (1) (2015) 1–24.
[31] M. Albano, M. Mordacchini, L. Ricci, AoI-Based multicast routing over voronoi overlays with minimal overhead, IEEE Access 8 (2020) 168611–168624.
[32] J. González Salinas, F. Boronat Seguí, A. Sapena Piera, J. Pastor Castillo, Key technologies for networked virtual environments, Multimedia Tools Appl. 82 (2023) 41471–41537.
[33] B. Yang, H. Garcia-Molina, Designing a super-peer network, in: U. Dayal, K. Ramamritham, T.M. Vijayaraman (Eds.), Proceedings of the 19th International Conference on Data Engineering, CoDE'03, IEEE, Bangalore, India, 2003, pp. 49–60.
[34] R. Süselbeck, G. Schiele, C. Becker, Peer-to-peer support for low-latency massively multiplayer online games in the cloud, in: Proceedings of 8th Workshop on Network and System Support for Games, NETGAMES'09, IEEE, Paris, France, 2009, pp. 1–2.
[35] T. Yoshihara, S. Fujita, Towards fog-assisted virtual reality MMOG with ultra-low latency, Int. J. Comput. Netw. Commun. 12 (6) (2020) 33–47.
[36] SamKnows, Quality of broadband services in the EU, 2014, European Commission Web Site.
[37] C. Jiang, A. Kundu, M. Claypool, Game player response times versus task dexterity and decision complexity, in: V.V. Abeele, M. Birk (Eds.), Proceedings of the Annual Symposium on Computer-Human Interaction in Play, CHI PLAY'20, ACM, Ottawa, Canada, 2020, 277–281.
[38] B. Nédelec, P. Molli, A. Mostefaoui, CRATE: Writing stories together with our browsers, in: I. Horrocks, B.Y. Zhao (Eds.), Proceedings of the 25th International Conference Companion on World Wide Web, WWW'16 Companion, ACM, Montréal, Québec, Canada, 2016, pp. 231–234.
[39] M. Nicolas, V. Elvinger, O. Gérald, C.L. Ignat, F. Charoy, MUTE: A peer-to-peer web-based real-time collaborative editor, in: Proceedings Od the 15th European Conference on Computer-Supported Cooperative Work—Panels, Posters and Demos, ECSCW'17, EUSSET, Sheffield, United Kingdom, 2017, pp. 1–4.
[40] H.-G. Roh, M. Jeon, J. Lee, Replicated abstract data types: Building blocks for collaborative applications, J. Parallel Distrib. Comput. 71 (3) (2011) 354–368.
[41] S.J. Castiñeira, A. Bieniusa, Collaborative offline web applications using conflict-free replicated data types, in: C. Baquero, M. Serafini (Eds.), Prooceedings of the 1st Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC'15, ACM, Bourdeaux, France, 2015, pp. 1–4.
[42] M. Mukund, K.N. Kumar, M. Sohoni, Bounded time-stamping in message-passing systems, Theoret. Comput. Sci. 290 (1) (2003) 221–239.
[43] M. Mukund, G. Shenoy R., S.P. Suresh, Optimized OR-sets without ordering constraints, in: M. Chatterjee, J. n. Cao, K. Kothapalli, S. Rajsbaum (Eds.), Proceedings of the 15th International Conference on Distributed Computing and Networking, ICDCN'14, in: Lecture Notes in Computer Science 8314, Springer, Coimbatore, India, 2014, pp. 227–241.
[44] J.B. Almeida, P.S. Almeida, C. Baquero, Bounded version vectors, in: R. Guerraoui (Ed.), Proceedings of the 18th International Symposium on Distributed Computing, DISC'04, in: Lecture Notes in Computer Science 3274, Springer, Amsterdam, The Netherlands, 2004, pp. 102–116.
[45] G. Kaki, S. Priya, K. Sivaramakrishnan, S. Jagannatha, Mergeable replicated data types, in: E. Visser (Ed.), Proceedings of the 34th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'19, ACM, Athens, Greece, 2019.
[46] R. Biswas, M. Emmi, C. Enea, On the complexity of checking consistency for replicated data types, in: I. Dillig, S. Tasiran (Eds.), Proceedings of the 31st International Conference on Computer Aided Verification, (CAV'19), in: Lecture Notes on Computer Science 11562, Springer, New York City, NY, USA, 2019, pp. 324–343.
[47] S.A. Weil, S.A. Brandt, E.L. Miller, C. Maltzahn, CRUSH: Controlled, scalable, decentralized placement of replicated data, in: B. Horner-Miller (Ed.), Proceedings of the ACM/IEEE Conference on Supercomputing, SC'06, ACM, Tampa, FL, USA, 2006.

[48] S. Gustavsson, S.F. Andler, Decentralized and continuous consistency management in distributed real-time databases with multiple writers of replicated data, in: L. Di Pippo, V. Kalogeraki (Eds.), Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'05, IEEE, Denver, CO, USA, 2005.
[49] M. Barbosa, B. Ferreira, J. Marques, B. Portela, N. Preguiça, Secure conflict-free replicated data types, in: S. Banerjee, T. Masuzawa (Eds.), Proceedings of the 22nd International Conference on Distributed Computing and Networking, ICDCN'21, ACM, Nara, Japan, 2021, pp. 6–15.
[50] B. Knutsson, H. Lu, W. Xu, B. Hopkins, Peer-to-peer support for massively multiplayer games, in: M. Krunz, B. Li, P. Mohapatra (Eds.), Proceedings of the 23rd Annual Conference on Computer and Communications, INFOCOMM'04, IEEE, Hong Kong, China, 2004, pp. 96–107.
[51] S. Yamamoto, Y. Murata, K. Yasumoto, M. Ito, A distributed event delivery method with load balancing for MMORPG, in: Proceedings of 4th Workshop on Network and System Support for Games, NETGAMES'05, ACM, Hawthorne, NY, USA, 2005.
[52] A. Yu, S.T. Vuong, MOPAR: A mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games, in: W.-C. Feng, K. Mayer-Patel (Eds.), Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV'05, ACM, Skamania, WA, USA, 2005, pp. 99–104.
[53] C. GauthierDickey, V. Lo, D. Zappala, Using N-trees for scalable event ordering in peer-to-peer games, in: W.-C. Feng, K. Mayer-Patel (Eds.), Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV'05, ACM, Skamania, WA, USA, 2005, pp. 87–92.
[54] T. Iimura, H. Hazeyama, Y. Kadobayashi, Zoned federation of game servers: A peer-to-peer approach to scalable multi-player online games, in: W.-C. Feng (Ed.), Proceedings of 3rd Workshop on Network and System Support for Games, NETGAMES'04, ACM, Portland, OR, USA, 2004, pp. 116–120.
[55] D. Frey, J. Royan, R. Piegay, A.-M. Kermarrec, E. Anceaume, F. Le Fessant, Solipsis: A decentralized architecture for virtual environments, in: Proceedings of the 1st International Workshop on Massively Multiuser Virtual Environments, MMVE'08, Reno, NV, USA, 2008.
[56] S. Holzapfel, S. Schuster, T. Weis, WoroStore—A secure and reliable data storage for peer-to-peer-based MMVEs, in: C. Badica, R. Menezes (Eds.), Proceedings of the 11th International Conference on Computer and Information Technology, CIT'11, IEEE, Paphos, Cyprus, 2011, 35–40.
[57] W.-C. Chang, P.-C. Wang, A write-operation-adaptable replication system for multiplayer cloud gaming, in: D. Tygar, C.-Y. Huang (Eds.), Proceedings of International Conference on Dependable and Secure Computing, DSC'17, IEEE, Taipei, Taiwan, 2017, pp. 334–339.
[58] B. Shen, W. Tan, J. Guo, P. Qin, B. Wang, An equity-based incentive mechanism for persistent virtual world content service, Serv. Oriented Comput. Appl. 14 (2020) 227–241.
[59] A. Bharambe, J. Pang, S. Seshan, Colyseus: A distributed architecture for online multiplayer games, in: L. Peterson, T. Roscoe (Eds.), Proceedings of the 3rd Conference on Networked Systems Design and Implementation, NSDI'06, USENIX, San José, CA, USA, 2006.
[60] E. Carlini, L. Ricci, M. Coppola, Integrating centralized and peer-to-peer architectures to support interest management in massively multiplayer online games, Concurr. Comput.: Pr. Exp. 27 (13) (2015) 3362–3382.
[61] A. Chandler, J. Finney, On the effects of loose causal consistency in mobile multiplayer games, in: Proceedings of 4th Workshop on Network and System Support for Games, NETGAMES'05, ACM, Hawthorne, NY, USA, 2005.
[62] B. Shen, J. Guo, Efficient peer-to-peer content sharing for learning in virtual worlds, J. Univers. Comput. Sci. 25 (5) (2019) 465–488.
[63] D.R. Jefferson, Virtual time, ACM Trans. Prog. Lang. Syst. 7 (3) (2005) 404–425.
[64] S. Ferretti, A synchronization protocol for supporting peer-to-peer multiplayer online games in overlay networks, in: R. Baldoni (Ed.), Proceedings of the 2nd International Conference on Distributed Event-Based Systems, DEBS'08, Rome, Italy, 2008, pp. 83–94.
[65] J. Xu, B.W. Wah, Consistent synchronization of action order with least noticeable delays in fast-paced multiplayer online games, ACM Trans. Multimed. Comput. Commun. Appl. 13 (1) (2016) 1–25.
[66] A. McCoy, T. Ward, S. McLoone, D. Delaney, Multistep-ahead neural-network predictors for network traffic reduction in distributed interactive applications, ACM Trans. Model. Comput. Simul. 17 (4) (2007).
[67] D. Hanawa, T. Yonekura, A proposal of dead reckoning protocol in distributed virtual environment based on the taylor expansion, in: D. Thalmann, A. Sourin (Eds.), Proceedings of the International Conference on Cyberworlds, CW'06, IEEE, Lausanne, Switzerland, 2006, pp. 107–114.
[68] Z. Zhang, T.E. Ward, S. McLoone, An information-based dynamic extrapolation model for networked virtual environments, ACM Trans. Multimed. Comput. Commun. Appl. 8 (3) (2012) 1–19.

Francesco Bertolotti is currently a Computer Science Postdoctoral Fellow Researcher at Università degli Studi di Milano and a member of the ADAPT Laboratory. Previously, he was a Computer Science Ph.D. student and a Research Assistant at the same University where he also got his master degree in Computer Science. His research interests are machine/deep learning techniques mainly focused to interpretability and understanding of sequence models. Previously, he worked on research topics related to exotic compilers, language development and deep learning applications to software engineering and development. More information about Dr. Bertolotti and all his publications are available at https://homes.di.unimi.it/bertolotti/. He can be contacted at francesco.bertolotti@unimi.it for any question.

Walter Cazzola is a Full Professor in the Department of Computer Science at the Università degli Studi di Milano, Italy, and Chair of the ADAPT laboratory. Dr. Cazzola is the designer of the mChaRM framework, @Java, [a]C#, and Blueprint programming languages, and is currently involved in the design and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has authored over 100 scientific publications. His research interests include software maintenance, evolution, and comprehension, as well as programming methodologies and languages. He has served on the program committees and editorial boards of major conferences and journals in his areas of expertise. He is also an Associate Editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and his publications is available at https://cazzola.di.unimi.it, and he can be contacted at cazzola@di.unimi.it for any inquiries.

Luca Favalli is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got his Ph.D. in computer science from the Università degli Studi di Milano. He is involved in the research activity of the ADAPT Lab and in the development of the Neverlang language workbench and of JavAdaptor. His main research interests are software design, software (and language) product lines and dynamic software updating with a focus on how they can be used to ease the learning of programming languages. He can be contacted at favalli@di.unimi.it for any question.

Dario Ostuni is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got is Ph.D. in computer science at the University of Verona. He received his M.Sc. in Computer Science at the University of Milan, where he was involved in the research activities of the ADAPT laboratory. His research interests concern algorithms and data structures, computational complexity and combinatorial optimization. He can be contacted at dario.ostuni@unimi.it for any inquiry.

Leonardo Secco has a B.Sc. in Computer Science from the Università degli Studi di Milano. He is a member of the ADAPT laboratory. His research interests are distributed systems, data structures and algorithms. He can be contacted at leonardo.secco@studenti.unimi.it for any question.