Distributing the Game State of Online Games: Towards an NDN Version of Minecraft

Philipp Moll, Sebastian Theuermann, Hermann Hellwagner
Institute of Information Technology
Klagenfurt University
{fistname}.{lastname}@aau.at

Jeff Burke
School of Theater, Film and Television
UCLA
jburke@remap.ucla.edu

***Abstract*—Online games nowadays play an undeniably important role in the entertainment industry. The continuously increasing popularity of these services goes hand in hand with increased complexity of technical challenges. The networking part of current games relies on decades-old technologies, which were never intended to be used for today's large-scale online games. Replacing the currently used connection-oriented networking approach by a content-centric architecture could yield advantages reaching beyond only avoiding inefficiencies found in IP-based online games. We propose a concept for a distributed Minecraft architecture, making use of these advantages by utilizing *Named Data Networking* (NDN) as the architectural basis. Our design decisions were guided by the insights we gained from examining Minecraft as a representative of current online games.**

I. Introduction

Gaming plays a remarkable role in the modern entertainment industry, becoming more important every year. According to the annual report of the Entertainment Software Association (ESA)[1], the US video game industry alone generated USD 36 billion in revenues in 2017 resulting from 2.6 billion people worldwide playing video games that year.

Besides the success in entertainment, games are a driving force behind the development of various hardware and software products. The possibility to interact with others in a shared, virtual environment in real-time is a well-known concept of online multi-player games, but is increasingly adopted by other applications as well. Using VR for multi-user conferencing in virtual environments [1] is only one example where technologies originating from video games are used for professional applications outside the entertainment sector.

Despite the popularity and importance of gaming, the networking part of computer games continues to rely on decades-old technologies. These technologies were never intended to be used for low-latency communication and efficient one-to-many information dissemination, which is part of the reason for overloaded game servers, especially at peak hours. When developing new applications based on gaming platforms, it is important that those platforms are built on a reliable base. This is one reason why we focus on replacing the current networking layer of games with a technology better suited to fulfill the requirements of modern gaming.

[1]https://www.esaannualreport.com/a-letter-from-michael-d.-gallagher.html, last accessed: 2019-01-31

Named Data Networking (NDN) [2] is a future Internet architecture with a strong focus on content instead of connections, and comes with advantages such as an inherent multicast functionality and security already implemented at the packet level. In NDN, a system-wide unique name is assigned to each piece of information. When requesting information, a so-called *Interest* packet carrying the name of the information is emitted and forwarded through the network based on the name, until a copy of the requested information is found. This information is encapsulated in a *Data* packet and sent back to the requester on the reverse path of the Interest.

When thinking about the distribution of the game state of online games as an information (or, content) distribution problem, where each part of the state is an individual piece of information, content-oriented approaches generally seem to be better suited than our current connection-oriented IP architecture. This is what motivates us to replace the networking part of the well-known game *Minecraft* by the content-centric NDN architecture. Apart from being representative for many online games, Minecraft has a highly modifiable game world and a richer feature set than many other online games. The content-centric nature of NDN could be used to split this modifiable world into small pieces, allowing to distribute the game state, which opens the possibility to form multi-server or peer-to-peer architectures. This paper represents the initial work on this endeavor and presents the basic concepts of this effort.

II. Related Work

The mismatch of the capabilities of IP-based architectures and the networking requirements of modern games is discussed and indicated by the simulation of network traffic of the popular *Battle Royale* game *Fortnite* in [3]. Chen et al. [4] argue that client-server architectures for games do not scale in IP; they present a decentralized content-oriented gaming architecture, which outperforms an IP-based architecture in simulation and emulation. Wang et al. [5] create a demonstration showing the practicability of NDN for games. They apply a sophisticated naming scheme to request information about objects in the proximity of a player.

III. What is Minecraft and How does it Work?

The game Minecraft is a 3D sandbox construction game with online multi-player capability. The game world is proce-

durally generated and players have the means to change every single part of the game world. The world, which is almost infinite in size, is built out of cubic blocks. Those blocks can be destroyed, or they can be placed by players to create buildings or all other imaginable structures. In addition to this building feature, the game mechanics allow to craft different types of items or to compete against monsters or other players.

Despite the unusual and intentionally simplistic graphics of Minecraft, the game is currently very popular. In 2018, 91 million users actively played Minecraft[2], and it ranges among the top 20 games on the live streaming video portal Twitch[3].

A. Minecraft as a Research Platform

One reason for the success of Minecraft is the vibrant community effort to improve the game, by means of modifications, and the permissive stance of Minecraft's publisher towards those modifications.

This openness allows to modify the network stack, which was already demonstrated by the work of Engelbrecht et al. [6], who created a distributed version of Minecraft based on TCP/IP. Besides the possibility to use Minecraft for research on networking, game elements and mechanics of Minecraft are well-documented[4] and an extensive protocol specification[5] exists, which eases research using the game.

Another reason for choosing Minecraft as a platform can be seen when focusing on the game elements in Minecraft, which are similar to those found in many other games. The possibility to permanently influence and change the game world is seen in many building games. The low-latency demands known from games of the shooter genre can be found in player-versus-player game modes supplied by community plugins for Minecraft as well. Also, when considering the possibility to extend the game to a peer-to-peer version, the fact that every player can change the complete world leads to important research challenges.

B. Minecraft Internals

When trying to improve a system like a game, it is important to understand the main components of the game first. For this work, we take Minecraft as a representative for online games, highlighting that many elements of games and the core concepts of online games are more or less the same for most state-of-the-art games. In this section, we dissect the Minecraft world into the smallest parts and describe the basic process of the game simulation.

1) Game World: In this paper, the umbrella term *object* summarizes all game elements of Minecraft, which are outlined in Fig. 1. *Blocks* are the structural components defining the game world in Minecraft. Individual blocks are defined by their material and additional block-specific data. The whole state of a block can be encoded in only a few bytes.

[2]https://www.gamesindustry.biz/articles/2018-10-02-minecraft-exceeds-90-million-monthly-active-users, last accessed: 2019-01-31

[3]https://www.twitch.tv/, last accessed: 2019-01-31

[4]https://minecraft.gamepedia.com/Minecraft_Wiki, last visited: 2019-01-31

[5]https://wiki.vg/Protocol, last accessed: 2019-01-31

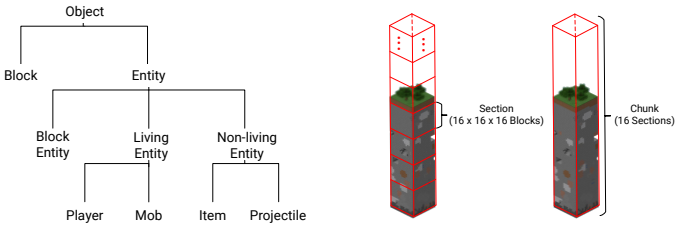

Fig. 1. Hierarchical structure of the objects of Minecraft's game state and organization of blocks in sections and chunks.[6]

Whereas blocks represent static structural components, *entities* represent dynamic objects. Entities can be classified into *block entities*, *living entities*, and *non-living entities*. A block entity can be seen as a complex block which allows user interaction. Entities which can freely wander around in the world can be seen as living entities. Living entities can be mobs, which are creatures controlled by the game server, or player avatars. In contrast to the almost unrestricted movement possibilities of living entities, the behavior of non-living entities can be easily predicted by simply applying physics. Items are one representative of non-living entities. An item lies around in the world after it is either dropped by a living entity or by harvesting a block. Projectiles, such as arrows, once created fly through the world until they hit an object.

The *game state* of Minecraft is represented by all objects which exist in the world. The structure of the world is defined by blocks; entities breathe life into the otherwise uneventful world. Each block in the Minecraft world can be identified by a coordinate triplet. However, for storage purposes, blocks are organized into groups of 16x16x16 blocks, which are referred to as *sections*. A *chunk* consists of 16 vertically stacked sections. The right part of Fig. 1 visualizes this structure.

The dynamic behavior of entities and the fundamental differences between different types of entities call for a flexible way of storage, which is why the concept of *Named Binary Tags* (NBTs)[7] is used. NBTs store entities and their attributes as binary strings in a tree-like representation.

2) Game Simulation: The game state needs to change over time. Games can be seen as simulations evolving in defined time intervals, referred to as *tick intervals*. Events occurring during a tick interval are applied to the game state and result in a new state after the tick interval has elapsed. In Minecraft, the tick interval is 50 ms, which means that 20 different game states are traversed each second.

In order to save computing resources, the simulation of Minecraft's almost infinite world is restricted to the parts of the world roughly enclosing the *Area of Interest* (AoI) of players. The world outside of this simulated area stays frozen, where the game state stops evolving until the area is vivified due to becoming part of a player's AoI again.

As typical for a client-server setting, the server manages the *primary copy* of the game state and distributes *replicas* of the

[6]Parts of the image are adopted from https://minecraft.gamepedia.com.

[7]https://minecraft.gamepedia.com/NBT_format, last accessed: 2019-02-01

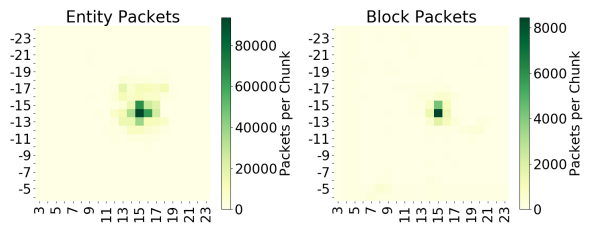

Fig. 2. Spatial distribution of the information carried by client-bound packets.

game state to all clients. When a client wants to control the player avatar, it sends a control command to the game server, which processes all commands received during a tick interval and thereby creates a new game state, which is distributed as replicas to all clients after the tick interval has elapsed.

IV. Analyzing IP-based Minecraft

Before building a system for gaming over NDN, the current state-of-the-art is analyzed, and it is evaluated which aspects of current systems can be improved. Therefore, we analyze how the internal game state of a Minecraft server changes over time and how data is sent to clients, by analyzing network traffic produced during a multi-player Minecraft session.

Besides reviewing the public documentation of Minecraft, a static code and protocol analysis was conducted, but also the dynamic behavior was analyzed by conducting an experiment including four persons playing Minecraft. The players in the experiment were instructed to build a *nether portal*. These game elements allow players to travel from the inital Minecraft world to the *nether* dimension, which is a hidden part of the world, but worth going to because of various factors. The players were further instructed to build the portal as a team and to form two groups to complete all sub-tasks required for building such a portal. This reflects common behavior for collaborative Minecraft playstyles. The experimental setup consisted of a dedicated Minecraft server (Minecraft version 1.12.2) and four laptops connected to the server via a local network. Encryption of Minecraft payload was disabled to allow for easier traffic analysis. The experiment resulted in a Minecraft network trace and an observation on how the server's internal game state changes over time, which are analyzed and discussed in the following.

A. Network Traffic of Minecraft

The network traffic during the game was recorded by using the network traffic analyzer TCPDUMP[8] on the server side. The Minecraft protocol specification was used to map TCP traffic to Minecraft packets and to extract game specific information like entity IDs or the geographic position of objects in the game world. This allows to distinguish between packets containing information about blocks and other structural components of the game world (referred to as *block packets*) and packets containing information about entities (referred to as *entity packets*).

[8]https://www.tcpdump.org/, last accessed: 2019-01-28

Focusing on map regions which were updated by network packets during the game, visualized in Figure 2, we can see that a large map region is covered by network packets, but the largest part of packets is focused on only a few chunks in the center of the covered region. The area covered by network packets ranges from chunk indices $X \in \{-32, ..., 8\}$ and $Z \in \{-9, ..., 28\}$, where X represents chunk indices on the east-west axis and Z indices on the north-south axis of the map.

When interpreting the figure and keeping the players' instructions in mind, we see that the individual player avatars are close-by on the map and assume that also the AoIs of the individual players overlap. A game state update happening in the AoI of a player means that a packet carrying the update has to be transmitted to the player. Overlapping AoIs mean that multiple players require the information about the same update. Because the current trace mostly carries information for a small map region only, we assume that benefits from multicasting information can be realized.

To quantify possible benefits achievable by using multicast, we mapped the information from entity packets to $(chunkId, tickNo, entId, pType)$ quadruples, where the $chunkId$ uniquely identifies the chunk where the game state update occurred, $tickNo$ represents the calculated tick number since the start of the game, $entId$ denotes the unique identifier of the entity causing the update, and $pType$ specifies the packet type according to the protocol specification.

A quadruple uniquely identifies a packet because a unique entity can perform the same action only once per tick. Therefore, duplicated quadruples identify packets carrying the same information redundantly to different clients. We count $2,646,065$ different entity packets, but only $1,004,554$ unique quadruples. Interpreting these amounts, we find that about 62% of all packets are sent redundantly and could be eliminated by utilizing an efficient multicasting system.

Further classifying entity packets by grouping them by type, we can find four different entity types in the network trace. The classified entity types are *players*, *mobs*, *items*, and *projectiles*. Reflecting the nature of these entity types, we know that living entities are far less restricted in what they can do compared to non-living entities. This can also be observed when focusing on the inter-packet intervals of the found entity types (Fig. 3).

As can be seen in Fig. 3, for arrows, the boxplot box – representing the center 50% of the data – is so narrow

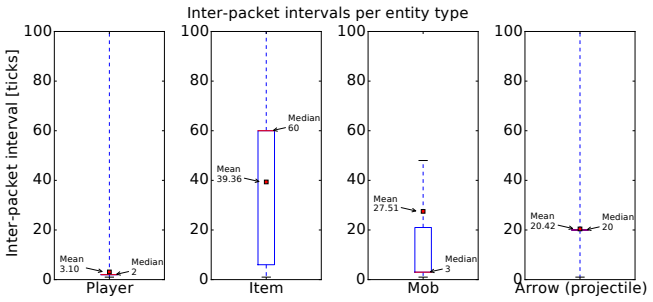

Fig. 3. Inter-packet intervals of entity packets reflecting the predictable behavior of different entity types.

that it even disappears in our representation, meaning that in most cases the interval between consecutive arrow packets is 20 ticks (1 second). This comes because it is not necessary to update the state of arrows more frequently, because the way arrows glide through the world is predictable. The inter-packet interval for items is less regular, but usually also quite large, most likely because the behavior of items is influenced by changes in the world structure (e.g., items falling down if the ground beneath them is destroyed). For mobs and players, however, the inter-packet interval is usually very small with a median of 2 for players and 3 for mobs, resulting from the unpredictable behavior of living entities. For every movement, updates need to be sent. The long upper whiskers most likely result from periods where objects left the AoI of a client and reentered later.

B. Tracing Changes in the Game State

A Minecraft server plugin was created to observe game state changes in the Minecraft world. The plugin creates snapshots of the game state at fixed intervals of 500 ms. By comparing two consecutive snapshots, changes of the game state can be calculated. The plugin labels game state changes with the current server time and stores them in log files, allowing to analyze the changes in a later post-processing step.

The comparison of game state snapshots results in a fine grained list of changes. The idea is to make the model of the game state changes as accurate as possible. By analyzing the change log, a lower and upper bound for the size of the game state changes between two consecutive snapshots can be calculated. The lower bound involves only the number of actually changed bytes without indication of where to apply the changes. For the upper bound, changes to a block or entity are always counted as if all properties of a block or entity would have changed (transmission of the whole block/entity), including information on where to apply the changes. Entity changes make up a large portion of the state change size since the full NBT representation of an entity is considerably larger than the full state of a single block. There were always at least 250 concurrently active entities during the recorded multi-player session. About 73% of all changes to entities affected living entities and 27% non-living entities. Fig 4 illustrates the lower and upper bounds calculated for the example Minecraft session. Interpreting the figure, it becomes noticeable that entity state changes are the primary contributor in terms of game state update size. Also, the amount of entity changes seems to stay roughly the same for longer time periods, while block changes tend to occur in bursts. For the lower bound, the mean size of the state changes is about 2.5 kB, while the mean size for the upper bound is about 140 kB. We expect the actual size of game state updates for a game server with a comparable number of simulated chunks and a similar update interval to be close to the lower bound.

The sum of changes over the whole session was about 14.7 MB for the lower bound and about 845 MB for the upper bound. Table I breaks down the total number of changes and relative size of all changes for blocks and entities in greater

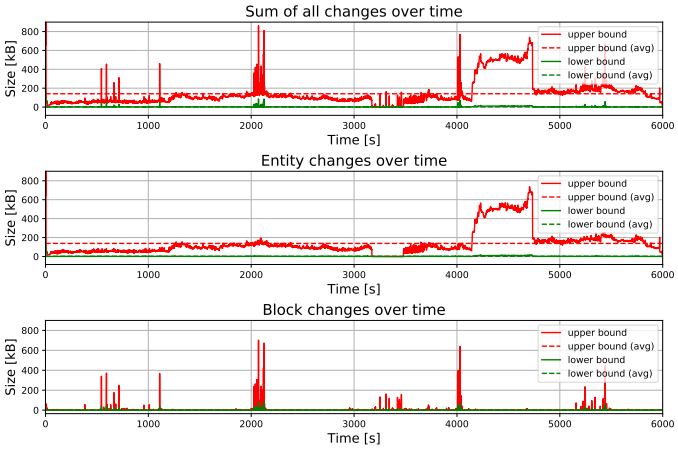

Fig. 4. Lower and upper bounds for the size of game state snapshot differences.

detail. Although the number of block changes is significantly larger than the number of entity changes, the collective size of block changes is small compared to the size of entity changes. So, while changes to blocks happen more often on average, their size is dwarfed by the size of entity changes.

	Number		Size (lower)		Size (upper)	
	abs.	%	MB	%	MB	%
Block changes	1,271,803	81.2	1.7	11.5	16.4	1.9
Entity changes	294,088	18.8	13.0	88.5	828.2	98.1
Total changes	1,565,891	100.0	14.7	100.0	844.6	100.0

TABLE I
SUMMARY OF BLOCK AND ENTITY CHANGES.

Throughout the gaming session, the number of chunks simulated by the server was around 500, except for a short duration with about 1000 simulated chunks, where the player avatars were separated in different dimensions of the game world. The median of the number of changed chunks per second was 30 (mean: 31.2, std. dev.: 13.4), which means that only a small fraction of all simulated chunks did change between consecutive snapshots. The mean number of changed sections per changed chunk was 1.15 (std. dev.: 0.19), meaning that usually only one section per chunk changes.

V. Towards an NDN Minecraft Version

In this section, the insights into state-of-the-art online games described in the last section are consolidated and used to design a distributed NDN-based version of Minecraft. By using the benefits of content-centric architectures, we not only intend to match the performance of connection-driven architectures, but also introduce new possibilities for the realization of a shared game world. The content-oriented paradigms for sharing the global game state increase the scalability of our proposed solution, but could be extended to a peer-to-peer solution as well. Besides that, our proposed system avoids inefficiencies observed in the client-server communication of the current Minecraft implementation.

The main components of the proposed architecture are the distribution of the game state in a server cluster and the connection of clients to the distributed game state via NDN.

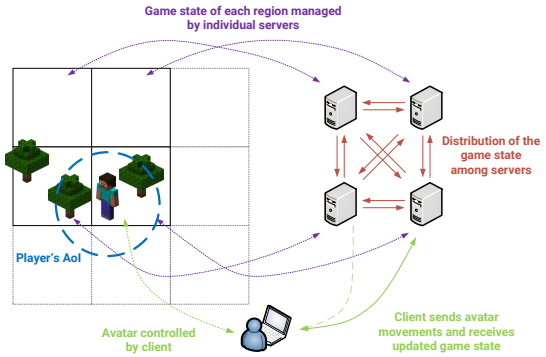

Fig. 5. Overview of the NDN-based Minecraft architecture.[6]

A high-level picture of the components and their interplay is given in Figure 5.

The distribution of Minecraft's game state is achieved by applying the zoning approach [7], where each server manages the *primary copy* of the game state for a defined map region. *Immutable replicas* of the server's primary copy are distributed to the other servers of the multi-server architecture. Finally, each server can than infer a global game state by using the received replicas. Details on the game state distribution can be found in Section V-A.

When connecting clients to the distributed game state using NDN, fulfilling low-latency demands while avoiding inefficiencies found in IP-based solutions are our primary goals. Therefore, latency-tolerant and latency-critical traffic is handled using different methods. Latency-critical game state updates in the AoI of the player are handled by the *Notify and Polling* approach, introduced in Section V-B. The distribution of structural information of the world, which is latency-tolerant information, can benefit from NDN's inherent multicast and in-network caching functionalities.

A. Distributing the Game State

The game world in Minecraft consists of independently managed chunks. This separation into comparatively small independent parts can be exploited by letting separate servers simulate different regions of the game world. The amount of work for an individual server would thereby be limited to managing the primary copy of the game state of all chunks in its region. In order to deduce a global game state for the whole world, servers then have to exchange game state information with each other.

Our proposed system beneficially uses this concept by utilizing a server cluster to distribute the computational effort arising by simulation of chunks, in order to increase the scalability of the overall system. The real complexity of this concept lies in the synchronization of the distributed game state. This becomes necessary during simulation when inter-dependencies between the individual chunks arise. Examples for such interdependencies are trees growing across chunk boundaries or living entities entering and leaving chunks.

Understanding the game state of individual chunks as content which changes over time, the game state synchronization among the nodes of the server cluster becomes a content distribution problem. By naming individual chunks and versioning the changes over time, the game state of each chunk can be separately requested, decoupling the data from the producing server. Decoupling the game state from the server allows for more flexibility on where to persist the state, resulting in higher reliability in case of server outage.

One way to implement the game state synchronization by using content distribution approaches is to use distributed dataset synchronization protocols, such as the NDN-based *PartialSync* (PSync) protocol [8]. PSync allows servers to subscribe to the game state managed by other servers in order to be notified when new state information is available. Once notified, the actual game state update can be retrieved by classical Interest/Data packet exchange. This approach supports a very fine-grained control of the state updates a server receives.

B. Connecting Clients to the Distributed Game State

This section describes the concepts used to connect clients to game states hosted by (potentially multiple) Minecraft servers via the NDN protocol. We first reformulate our observations of current IP-based systems and infer efficient NDN-based solutions for tackling the observed inefficiencies. For connecting the clients, latency-tolerant traffic and latency-critical traffic needs to be handled differently. Latency-tolerant traffic is intended for distributing structural information of the world, the latter one for game state updates in the AoI of clients.

1) Distributing Structural Information of the World: In the current IP-based implementation, the Minecraft server pushes structural information in the form of encoded chunk data to the client. Those encoded chunks can be up to a few kilobytes in size. Considering that structural information often does not change for long periods, sending this heavy type of data through the network whenever required by the client is inefficient, which is why using in-network caching, but also multicast could result in benefits.

When bringing this procedure to NDN, two issues need to be addressed. First, the current situation should be improved and only encapsulating information in NDN packets does not utilize the full potential of NDN. Second, NDN as a pull-based system does not support pushing data from the server to a client, as it is done in IP.

To enable multicasting and in-network caching of chunk data, a potential naming scheme is outlined. The geographic position of a chunk, which uniquely identifies it, is used as a name component. To be able to distinguish between the newest version and outdated versions of a chunk, an additional version field, which is increased whenever the structural information of the chunk changes, is added to the name. Resulting names could be structured as follows:

`/<app-pfx>/<game-pfx>/<chunk-loc>/<chunk-ver>`

The *app-pfx* component specifies Minecraft as the application and the *game-pfx* the target game instance. The *chunk-*

loc component uniquely specifies a chunk of the world, while *chunk-ver* is used to address a specific version of the chunk.

Instead of pushing structural information to clients, the server only notifies clients about which chunks to retrieve. For retrieving the chunks, classical Interest/Data exchange is used. Applying this concept, only the small notification messages are unicasted to relevant clients. The use of classical Interest/Data leads to the automatic use of NDN's built-in multicasting and in-network caching functionalities.

2) Latency-critical Game State Updates: Game state updates in the AoI of the client can be seen as latency-critical information, which is usually pushed through the network to keep the latency as low as possible. As push functionality is not directly available in NDN, we use *Notify and Polling* to achieve push-like behavior for parts of the information.

As analyzed in Section IV, the largest part of the game state changes is caused by entity updates. The traffic carrying updates for different entity types shows rather distinctive patterns. In general, living entities produce data every two to 20 ticks, whereas non-living entities produce data in larger intervals ranging between 10 and 60 ticks[9]. This allows clients to poll game state updates of entities by using classical Interest/Data exchange, where the interval between polling updates is specified by the update rate of the entity. The length of this polling interval can be adapted by simple heuristics, to better meet the current demand.

If an entity stops producing data or moves out of the client's AoI, polling is not meaningful anymore and stops in a defined way. If the entity starts moving again or moves back into the AoI of the client, the polling process needs to be restarted. Therefore, the server notifies all interested clients by multicasting a notification. This notification can be realized by names utilizing chunk locations as name components. The client's AoI can be represented as set of chunks around the player avatar. If a client wants to be notified about changes in a chunk, it emits a long-living Interest requesting notifications for that chunk. As soon as the necessity for a notification occurs, the server sends the notification as a classical NDN Data packet as response to the pending Interest. Thereby, the notification is automatically broadcasted to all relevant clients. An increasing sequence number as last part of the name of a notification can further be used for loss detection and recovery. After receiving the notification, the client can re-/start the polling process for the relevant entity. This interplay of notifying clients and polling game state updates is referred to as *Notify and Polling* approach.

VI. Conclusion

In this paper, we analyzed the internals of Minecraft as a representative for state-of-the-art online games and showed inefficiencies common to IP-based games. We proposed a concept for a distributed, NDN-based version of Minecraft, which distributes the game state among multiple servers in order to improve scalability. Furthermore, the proposed concept avoids inefficiencies found in IP-based implementations by utilizing NDN's inherent multicasting and in-network caching functionalities. The proposed concepts can be used to manage the distributed game state in a multi-server architecture as well as to connect clients to a distributed game state.

In order to evaluate the suitability of the concepts and to refine them, the next step is to realize them in the form of real-world prototypes. We already began extending the existing Minecraft implementation with functionality required for distributing the game state over multiple servers and building a proxy to connect the standard Minecraft client to this cluster over NDN, as proposed in [9].

The implementation and evaluation of the concepts discussed in this paper is only the first step towards utilizing the full spectrum of NDN-based solutions to improve the networking part of online games. Besides reducing inefficiencies, NDN could help in other aspects, such as increasing reliability as well. While recovering from server failures is only little effort in content-centric architectures, a self-organizing cluster, formed via NDN sync protocols such as *VectorSync* [10], could optimize resource usage, e.g., by dynamically splitting high-activity map regions to multiple servers or by merging idle regions and thereby increase player QoE and reduce the usage of server resources at the same time. Extending this idea could result in the full distribution of Minecraft's game state in the form of a peer-to-peer system.

Software artifacts and traces originating from this work are published as open source artifacts on Github (https://github.com/phylib/MinecraftNDN-RAFNET19).

[9]Sharper interval bounds can be specified when focusing on individual entity types.

References

[1] S. N. B. Gunkel, H. M. Stokking, M. J. Prins, N. van der Stap, F. B. t. Haar, and O. A. Niamut, "Virtual Reality Conferencing: Multi-user Immersive VR Experiences on the Web," in *Proc. 9th ACM Multimedia Systems Conference (MMSys)*, 2018, pp. 498–501.

[2] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, kc Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 66–73, July 2014.

[3] P. Moll, M. Lux, S. Theuermann, and H. Hellwagner, "A Network Traffic and Player Movement Model to Improve Networking for Competitive Online Games," in *Proc. 16th Annual Workshop on Network and Systems Support for Games (NetGames)*, 2018, pp. 1:1–1:6.

[4] J. Chen, M. Arumaithurai, X. Fu, and K. Ramakrishnan, "G-COPSS: A Content Centric Communication Infrastructure for Gaming Applications," in *Proc. IEEE 32nd Int'l. Conference on Distributed Computing Systems (ICDCS)*, 2012, pp. 355–365.

[5] Z. Wang, Z. Qu, and J. Burke, "Matryoshka: Design of NDN Multiplayer Online Game," in *Proc. 1st International Conference on Information-Centric Networking (ICN)*, 2014, pp. 209–210.

[6] H. A. Engelbrecht and G. Schiele, "Transforming Minecraft into a Research Platform," in *Proc. IEEE 11th Consumer Communications and Networking Conference (CCNC)*, 2014, pp. 257–262.

[7] A. Yahyavi and B. Kemme, "Peer-to-Peer Architectures for Massively Multiplayer Online Games," *ACM Computing Surveys*, vol. 46, no. 1, pp. 1–51, Oct. 2013.

[8] M. Zhang, V. Lehman, and L. Wang, "PartialSync: Efficient Synchronization of a Partial Namespace in NDN," NDN, Tech. Rep. NDN-0039, 2016.

[9] T. Liang, J. Pan, and B. Zhang, "NDNizing Existing Applications: Research Issues and Experiences," in *Proc. 5th ACM Conference on Information-Centric Networking (ICN)*, 2018.

[10] W. Shang, A. Afanasyev, and L. Zhang, "VectorSync: Distributed Dataset Synchronization over Named Data Networking," NDN, Tech. Rep. NDN-0056, 2018.