Inter-Server Game State Synchronization using Named Data Networking

Philipp Moll, Sebastian Theuermann, Natascha Rauscher, Hermann Hellwagner
Alpen-Adria-Universität Klagenfurt
{firstname}.{lastname}@aau.at

Jeff Burke
UCLA
jburke@remap.ucla.edu

ABSTRACT

In this paper, we develop a system for inter-server game state synchronization using the NDN architecture. We use Minecraft as a real-world example of online games and extend Minecraft's single-server architecture to work as multi-server game. In our prototype, we use two different NDN-based approaches for the dissemination of game state updates in server clusters. In a naive approach, servers request game state updates for small segments of the game world from other servers of the cluster. In an improved approach – the *region manifest approach* – servers identify changed parts of the world by subscribing to manifest files containing information about world regions managed by the other servers of the cluster. An apparent downside of the NDN approaches is the high overhead when handling small-sized game state updates, but our evaluation shows that NDN already improves on IP-based implementations regarding the resulting traffic volume when three or more servers are involved. Furthermore, caused by NDN's inherent multicast functionality, the advantage over IP increases with the size of the server cluster. Moreover, the use of NDN-based approaches leads to benefits beyond traffic reduction only. The name-based host-independent access to world regions allows to scale server clusters easier.

CCS CONCEPTS

• **Software and its engineering** → **Interactive games**; • **Networks** → *Network architectures.*

KEYWORDS

Multiserver Online Games, Named Data Networking, Game State Synchronization

ACM Reference Format:
Philipp Moll, Sebastian Theuermann, Natascha Rauscher, Hermann Hellwagner and Jeff Burke. 2019. Inter-Server Game State Synchronization using Named Data Networking. In *6th ACM Conference on Information-Centric Networking (ICN '19), September 24–26, 2019, Macao, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357150.3357399

ICN '19, September 24–26, 2019, Macao, China

ACM ISBN 978-1-4503-6970-1/19/09.
https://doi.org/10.1145/3357150.3357399

1 INTRODUCTION

During the last decades, the popularity of computer games increased steadily and now computer games represent a fundamental part of the entertainment industry. Video game sale revenues reached 43.4 billion USD in the U.S. only in 2018 [2]. Although many games are played online, the networking part of online games usually relies on decade old technologies, which were never intended to be used for gaming and are often part of the cause of overloaded and crashing game servers during peak hours.

Massive Multiplayer Online Role-Playing Games (MMORPGs) allow up to thousands of players to play in the same shared virtual world. Those worlds are often distributed on multiple servers of a server cluster, because a single server would not be able to handle the computational load caused by the large number of players interacting in a huge virtual world. This distribution of the world on a server cluster requires to synchronize relevant game state information among the servers. The synchronization requires every server to send updated game state information to the other servers in the cluster, resulting in redundantly sent traffic when utilizing our current IP infrastructure. We assume that efficient multicasting solutions could reduce the amount of redundant traffic. Novel information-centric networking (ICN) approaches inherently support network-level multicast and enable applications to decouple the game state information from the server producing it. This independence of state from the hosting server leads to additional advantages reaching beyond simply reducing inefficiencies such as redundantly sent traffic. For instance, recovery after server failures or also the possibility to dynamically add or remove game server instances from the cluster becomes easier when the game state is associated with a name instead of a specific server instance.

In this paper, we attempt to reduce the inefficiencies of IP-based inter-server game state synchronization by replacing it with one utilizing Named Data Networking (NDN) [19], a promising ICN architecture. We use the open-world game Minecraft to build a prototype and to implement multiple protocols to synchronize game state in a server cluster. This enables us to compare performance metrics of IP- and NDN-based inter-server game state synchronization approaches, but also to connect real-world game clients to the prototype and perceive the different synchronization alternatives from a player's perspective.

2 RELATED WORK

The inability of IP-based architectures to efficiently fulfill the networking requirements of modern games is discussed and indicated by the simulation of network traffic of the popular *Battle Royale* game *Fortnite* in [11]. In [12], we analyzed changes in Minecraft's

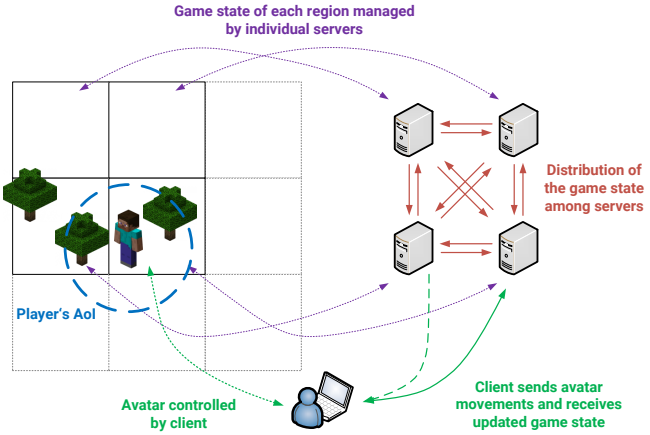

Figure 1: Overview of the envisioned NDN-based distributed gaming architecture.

game state as well as the network traffic produced during multiplayer gaming sessions and found that a large share of traffic is sent redundantly. It is assumed that the use of an ICN approach could efficiently eliminate redundant transmissions caused by IP's host-based nature. Chen et al. [3] argue that client-server architectures for games do not scale in IP; they present a decentralized, content-oriented publish/subscribe architecture for the transport of game data, which outperforms an IP-based architecture in simulation and emulation. Wang et al. [17] 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.

Engelbrecht et al. [5] try to improve networking of IP-based online games and demonstrate Minecraft as a research platform for online games. They further show that Minecraft can be adapted to work as a distributed server architecture.

3 ENVISIONED ARCHITECTURE

Our information-centric architecture for Minecraft includes a client-server communication and an inter-server synchronization approach, as visualized in Fig. 1. In this paper we focus on the latter, using ICN for the synchronization of servers supporting existing IP clients. The client-server approach motivates our approach to this synchronization.

Connection-oriented paradigms require clients to connect to a specific endpoint hosting the required game information. For online games hosted by a server cluster, this means that a client connects to a single server, which sends the client all information it needs. In fact, this single server has no direct access to objects located in regions of the game world handled by other servers, referred to as *remote regions*. This information is only relayed to the server via inter-server game state synchronization, resulting in higher latency (respectively stale data) when sending information about remote regions to clients.

3.1 Game State Synchronization

Most MMORPGs distribute the game world to a server cluster to overcome a shortage of computational resources. Every server of the cluster is exclusively managing a specific region of the game world. Together, the servers of the cluster manage the whole game world. Thereby the computational load of an individual server instance is reduced to only managing clients whose avatars are in the server's region. The downside of this distribution is the requirement to synchronize the game state of the diverse regions among the servers. This is necessary because the border of a server's region should not be noticeable for clients.

For synchronization purposes, a server splits all objects in the game world into primary copies and immutable replicas [18]. Objects in a server's own region are handled as primary copies, all objects in remote regions are handled as immutable replicas. A server has the exclusive right to change primary copies of objects and distributes the state of these objects to the other servers of the cluster. The state of received objects in remote regions, handled as immutable replicas, is shown to clients, but a server is not allowed to change the state of those replicas.

Synchronizing the state of the game objects means that the information needs to be transferred from the holder of the primary copy to all other servers of the cluster. In IP-based networks, the information is redundantly unicasted to every server. In ICN approaches, information about game objects gets published to be requested instead of being sent over connections to individual receivers. This principle shows a high potential to reduce redundant traffic, when implemented efficiently. One way to implement inter-server game state synchronization using ICN protocols is discussed in Section 5.

3.2 Client Communication and Scalability

An additional motivation for this work – yet not dealt with in this paper – is client communication using ICN and the simplified task of scaling server clusters. In ICN environments, there is no need for clients to connect to a specific server instance. Instead, clients issue requests for the information they need, and no matter which server hosts the information, the information is delivered. This results in lower latency, but also in higher availability, because not only a single server, but every networking component having a copy of the information is able to answer client requests.

Further, online games underlie continuous load changes caused by the game taking its course. Player avatars moving through the game world and traversing regions managed by different servers could lead to situations where many player avatars are in the same region. The result is heavy load for a single server while other servers are almost idle. Moreover, the client's ability to join or leave a game at any time results in the number of active players and thereby the load for the entire server cluster varying over time. Decoupling game state from the servers managing it allows easier handling of such load variations. A region not being bound to a specific server instance allows to move the management of the region from one server to another and thereby to dynamically adapt the region sizes of servers. The possibility to adapt server regions and responsibilities increases the cluster's scalability and allows to meet the continuously changing demands of the game.

4 THE MINECRAFT GAME

Minecraft is a 3D sandbox construction game with online multiplayer capability. The game world is procedurally generated and

players have the means to change every single part of the world. The game world itself is almost infinite in size and built out of cubic blocks. These blocks can be destroyed, or they can be placed by players to create buildings or other 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.

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. [5]. Besides the possibility to use Minecraft for research on networking, game elements and mechanics of Minecraft are well-documented [7] and an extensive protocol specification [8] exists, which eases research using the game.

4.1 Minecraft's Game State

Before discussing how online games can be improved, we briefly describe how they communicate. We take Minecraft as a representative for online games in general, because it contains many core concepts of state-of-the-art online games.

A game world in Minecraft is built out of cubic *blocks*; *entities*, such as player avatars and monsters, breathe life into the otherwise uneventful world. The *game state* of Minecraft consists of the state of all blocks and entities existing in a game world. Each block can be uniquely identified by a coordinate triplet. For more efficient storage, blocks are organized into groups of 16x16x16 blocks, referred to as *sections*. 16 vertically aligned sections make up a *chunk*. These chunks are used for storing the structure of the game world and for transferring it to clients.

4.2 Game Simulation

The game state in computer games changes over time. Games can be seen as simulations evolving in defined time intervals, referred to as *tick intervals*. Events occurring during a tick interval, such as blocks being broken or monsters moving, 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, meaning 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 in a frozen state, where the game state stops evolving until the area is vivified due to becoming part of a player's AoI again.

5 GAME STATE SYNCHRONIZATION

In distributed online games, the servers of a server cluster cooperatively simulate the progression of the game world's game state. In order to build a single consistent game state among all servers of the cluster, the changes to the game state – the *game state updates* – need to be synchronized among the server instances. Having knowledge of the game state of remote regions is necessary because events happening in a region can influence neighboring regions. So for instance, a growing tree can stretch its branches across the border of a region, or entities can walk from one region to another.

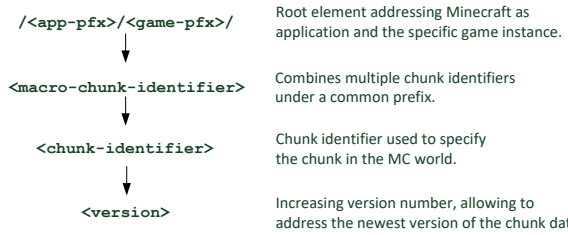

Figure 2: Namespace using the macro chunk identifier as hierarchical representation of the game world.

Also, when clients are connected to the server, the server needs to inform the client what the world looks like and which entities are in their view distance, even if the view distance extends beyond the border of the server's own region.

Building this single consistent game state is achieved by inter-server game state synchronization. Game state updates of each server's region are created and delivered to the other server instances. Each server can then infer the single consistent game state.

The most important requirement of game state synchronization is consistency, meaning that all servers of the cluster need to have the same view of the world. An inconsistent game state leads to the game world looking different on different servers of the cluster. For players this could mean that blocks or entities are suddenly appearing or disappearing when the player avatar crosses the border of a region and would negatively influence user satisfaction.

5.1 Naming the Game Components

In NDN, each piece of information is assigned a hierarchical, system wide unique name. We designed our namespace to facilitate efficient forwarding regarding the number of required entries in the NDN forwarding nodes' *Forwarding Information Bases* (FIBs). Utilizing the fact that NDN Interests are forwarded using longest-prefix matching, our namespace summarizes multiple geographically close chunks under a common prefix. Therefore, we add the *macro chunk identifier* as hierarchical representation of the world's structure. Instead of registering each chunk in the FIB, only the macro chunk identifiers summarizing chunks handled by the same server need to be registered. This leads to a significant reduction of the required FIB entries for a Minecraft game.

Our namespace design for inter-server game state synchronization is visualized in Fig. 2. The `app-pfx` component specifies Minecraft as application, while the `game-pfx` component uniquely identifies the targeted game instance. A chunk in Minecraft contains the full game state of a 16x16 block wide area of the game world. To infer the game state of the whole game, the game state updates of every single chunk need to be retrieved. For this purpose, the *chunk identifier*, consisting of its X and Z coordinates, is included in the NDN name.

A `version` field is used as the last name component to represent the continuous change of the game state. The version of a chunk increases for every new game state update. Thereby, it becomes possible to request a specific, respectively the latest version of a chunk's game state.

5.2 Synchronization and Distribution

User interaction with objects in a chunk results in changes to the chunk's game state, which need to be published as a game state update. The generation of such updates depends on user behavior and is therefore hard to predict. The central task of inter-server game state synchronization is to communicate the latest game state updates of the world chunks in a server's region to the other servers. We present two approaches for this update distribution task: a naive approach with direct chunk-level update retrieval and our region manifest approach (RMA), where information about the latest world state is distributed via region-specific manifest files.

Naive approach: The naive solution to implement inter-server game state synchronization is that every server requests the current game state updates on a chunk-level granularity directly from the remote servers by emitting Interests for every remote chunk. Since large parts of the game world change very infrequently and the change interval cannot be predicted, Interests for game state updates need to be emitted in regular intervals and would cause a high number of expiring Interests because no changes have happened. *Long-lived Interests* (LLIs)[1] could mitigate this issue. The idea is to issue LLIs for data items with unknown time of generation. When the LLI reaches the potential producer, it stays pending until the data is produced or the LLI's lifetime expires. When the data item is produced, the Data packet can immediately be sent to the requester as response to the LLI. In case of an LLI timing out, the requester needs to re-issue the LLI. In our naive approach servers issue LLIs for all world chunks in remote regions. When a chunk changes, an update is produced and immediately sent as reply to the pending LLI. When the update is received, an LLI for the next update (incremented version field) is emitted. One downside of this approach is the overhead resulting from issuing Interests for chunks which change infrequently. The low update rate of those chunks leads to the LLIs tending to time out without initiating the transmission of a game state update.

Critically, in NDN pending Interests are soft state; they are subject to link loss and are not guaranteed to persist in a node for their specified lifetime. So, not only does the increased lifetime of LLIs increase resource demands on all networking devices for maintaining the soft-state of pending LLIs, but performance on loss-prone links and resource-limited nodes could be limited. While loss can be mitigated by retransmissions of expired Interests when using Interests with a short lifetime, the long lifetime of LLIs directly translates to a longer time until packet loss is recognized.

Region manifest approach: Our region manifest approach (RMA) tackles the downsides of the naive approach by using a concept similar to NDN-based distributed dataset synchronization protocols (referred to as sync protocols). Sync protocols distribute manifest files representing the current state of the distributed dataset. For instance *ChronoSync* [21] uses *DigestTrees* to maintain dataset changes, *PSync* [20] utilizes *Invertible Bloom Filters* to represent the latest version of data items, and *VectorSync* [16] encodes the latest version of data items in *State Vectors*. The result of using a sync protocol is the knowledge of the dataset's state, but not the synchronized data itself.

[1] LLIs are Interests with lifetime set to a value close to the generation delay of the data they are requesting. We employ the term when that delay likely exceeds several RTTs - usually longer than a few seconds.

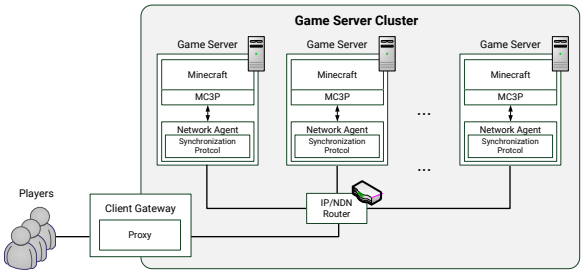

Figure 3: Prototype of our game server cluster.

In RMA, each server generates manifest files containing the latest versions of the chunks in its own region in constant intervals. Those manifest files are published under server-specific sync prefixes and contain information about the server's region size as well as the chunk versions of the described region as one-dimensional version vector. Other servers request the manifest file by Interest/Data exchange and use the received chunk version information to identify changed chunks by comparing the chunk versions of their local replicas to the versions published in the manifest file. Game state updates of changed chunks are then requested by Interest/Data exchange. This leads to a synchronization latency of roughly 1.5 RTT, where the delivery of the manifest file requires 0.5 RTT (the manifest can be pre-requested due to constant generation intervals) and fetching the game state updates takes 1.0 RTT.

Comparing RMA to sync protocols, the main difference is that in sync protocols multiple parties may publish data, while in RMA only a single producer is publishing data for a specific region. Further differences are the format and the distribution of the manifest file. While manifest files in ChronoSync and PSync are customized for each participant, RMA uses one manifest file for all clients yielding multicasting benefits for manifest file distribution. Unlike ChronoSync and PSync, the manifest distribution is based on Interest/Data exchange in RMA. ChronoSync and PSync both operate with LLIs, which might reduce the number of required Interests, but causes overhead for maintaining a soft state for pending Interests.

6 PROTOTYPE AND EVALUATION

In this section we describe our prototype of a distributed version of Minecraft and evaluate inter-server game state synchronization. The prototype mimics a server cluster hosting an online game in a data center. In the evaluation, we compare our NDN-based approaches, as described in the previous section, with an IP-based baseline implementation.

6.1 Multiserver Minecraft Prototype

The components of our prototype and their interplay are shown in Fig. 3. The prototype is based on the customizable Minecraft server implementation *SpigotMC* [15], our Minecraft Cluster Connector Plugin (MC3P) which detects and encodes game state updates for chunks in the region of the server, and a network agent synchronizing those updates with the other servers of the cluster.

A topology file informs all components of the prototype about the region they are managing as well as the regions that are managed by

other servers. The game simulation on a Minecraft server is limited to the region it is managing. The MC3P plugin takes snapshots of the region's game state in regular intervals, converts them to chunk-based game state updates and hands them over to the network agent which publishes these updates for the other servers.

Publishing the full game state in every update would lead to redundant transmission of information in subsequent packets. This is why our prototype differentiates between two different game state update types. An *update bundle* summarizes all changes to the game state of a chunk since the start of the game and can be seen as a checkpoint, because receiving a single update bundle allows to reconstruct the current game state. *Delta updates* encode only the game state changes since the last update (delta update or update bundle); applying delta updates is only meaningful if all delta updates since the last checkpoint (update bundle or start of the game) are present. We decided to combine the use of update bundles and delta updates for inter-server game state synchronization. This combines the advantages of small update sizes during normal operation and easier consistency recovery after server faults.

The network agents connected to the Minecraft servers are responsible for the actual inter-server game state synchronization. The protocol to be used by the network agent is easily configurable; currently a push-based IP approach, our naive NDN approach, and RMA are implemented.

A proxy server is used as a gateway, allowing standard Minecraft clients to connect to the distributed version of Minecraft. The proxy automatically connects the client to the server which manages the map region its avatar is located in and migrates the client to another server if the player avatar moves to a new region.

6.2 Evaluation Setup

As described in Section 3, we expect that an information-centric inter-server synchronization can reduce inefficiencies found in IP resulting from redundant unicasts. In this section, we describe a network emulation using the network emulator *Mini-NDN* [13] to measure the potential traffic reduction when using NDN.

The network topology of our emulated server cluster is sketched in Fig. 3. We emulate a single server environment as well as a server cluster in a data center with the number of servers hosting the distributed Minecraft game varying from two to four. A central NDN-enabled forwarder is connecting all servers of the cluster. An additional gateway node connected to the central forwarder is hosting the client proxy and a varying number of Minecraft clients. In order to emulate realistic player behavior, up to 20 players are emulated with the *Mineflayer* client emulator framework [14] and are configured to follow movement traces of the online multiplayer game Fortnite, published in [11]. The clients start connecting to the cluster after initialization of the Minecraft servers and disconnect when they reach the last waypoint of their movement trace. The length of the traces varies from a few seconds for some players up to over 15 minutes for others. In addition to the clients walking around, clients change the structure of the world by placing blocks when reaching waypoints, and additional movement arises by monsters wandering through the world.

The Minecraft server cluster is configured to host a flat world, of which a 96x96 chunks (1536x1536 blocks) large area is managed by

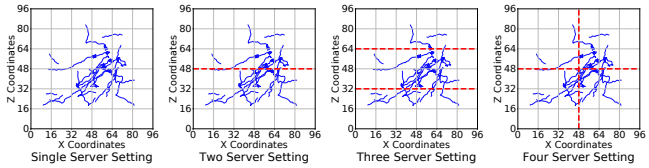

Figure 4: Emulated client movements and structure of regions with varying server cluster size.

the servers. The equally sized regions managed by the individual servers are depicted in Fig. 4. Chunk changes of 10 subsequent tick intervals (500 ms) are combined in a single game state update. Every tenth game state update of a chunk is realized as update bundle.

In the IP setting, servers push game state updates to all other servers of the cluster via TCP. For that, the ZeroMQ library [4] in publish/subscribe mode, where updates are pushed via TCP unicast messages to all subscribers, is used. In the evaluation, each server is publishing changes in its own region and is subscribing to the changes of all other servers in the cluster.

For NDN emulations, the NDN Forwarding Daemon (NFD) [1] in version 0.6.5 is running on all nodes in default configuration[2]. Names are realized as described in Section 5. Macro chunk identifiers combine 16x16 chunks in an additional hierarchy level, resulting in the managed world being covered by 6x6 macro chunks. For the naive approach, LLIs have a lifetime uniformly distributed between 60 and 120 seconds to prevent overloaded links caused by Interest timeouts. For RMA, the lifetime of Interests is set to 500 ms, which represents 10 tick intervals and manifest files are compressed with Gzip [9] in fast mode.

The main metric for comparison is the amount of network traffic produced by game state synchronization. Therefore, network traffic on all server nodes is captured. We decided not to focus on synchronization latency in this evaluation since the low link latencies in the server cluster scenario might not influence user behavior without inducing additional challenges for the network, such as latency variations or loss.

6.3 Results

The results of our evaluation are visualized in Fig. 5. Traffic volumes were acquired from the producer perspective. Visualized numbers for NDN include incoming Interests and outgoing Data on all server nodes; for IP, outgoing sync traffic on server nodes is shown.

The results indicate clear advantages of IP in the two server setting, where no benefits of multicasting are possible. NDN performance is impaired by the overhead introduced by Interests as well as by the larger protocol headers as compared to IP.

In the settings with larger server clusters, the advantages of NDN come to bear. In the three server setting, as a result of NDN's inherent multicast support, we see that the number of Data packets is already lower for both NDN approaches as compared to the IP implementation. However, the total number of sent packets is greater than in IP, caused by NDN's one-Interest-per-Data principle. Considering the naive approach, a high number of Interests are

[2]NFD configuration used for evaluation: forwarding strategy: Best Route; content store size: 65.536; cache policy: LRU; face protocol: UDP.

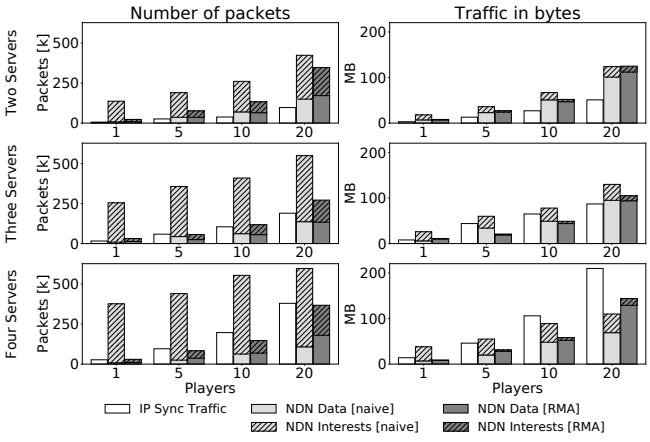

Figure 5: Synchronization traffic on server nodes with varying number of players and server cluster sizes.

timing out, leading to an even higher number of packets. Despite the high number of packets, when focusing on the number of sent bytes in the three server setting, we already see that RMA generates less traffic than IP. Only in the 20 player evaluation, the amount of traffic is higher in NDN, caused by the high amount of very small game state updates, penalizing NDN traffic with a relatively high overhead. With 20 players, we observed that only 37% of all captured bytes for the naive approach and 47% for RMA carry payload. The rest of the traffic results from Interest packets and Data packet header fields, such as the signature.

Increasing the number of servers, and thereby the number of required connections for the IP-based implementation, the advantages of NDN become clearer. In the four server setting, the traffic reduction resulting from multicasts outmatches the overhead induced by Interests and the larger NDN headers. RMA outperforms the IP version both in number of packets and total traffic volume.

6.4 Discussion

The results show that the inherent multicasting functionality of NDN helps reduce the traffic resulting from inter-server game state synchronization, especially in larger server clusters. Our results confirm our initial assumption that NDN's multicasting advantages become clearer at larger server clusters. It was observed that the traffic reduction was significantly larger in the four server setting than in the three server setting. We assume larger clusters to benefit even more, as unicast IP traffic volume is expected to increase quadratically with the number of server cluster participants, while NDN traffic volume is expected to increase linearly.

Even though a server cluster topology would allow for using network level multicast in IP, we point out that reliable multicasts in IP are not inherently supported [6] and would require additional middleware. In NDN, reliable multicast is easier to implement, since consumers simply need to re-issue timed-out Interests in the same way as they need to do in reliable unicast communication. Hence, NDN multicast is functioning in non-data center environments as well, such as in server clusters distributed over a continent.

With respect to the results of the NDN approaches, we see that RMA results in less traffic than the naive approach regarding the number of packets and the total traffic volume in bytes. The amount of Data packets is roughly the same for the naive approach and for RMA, which is expected because the same game state updates are generated and distributed in both approaches. Regarding the number of Interests, RMA only emits Interests for game state updates of changed chunks resulting in about the same number of issued Interests as sent Data packets. This number is worse in the naive approach, where Interests for every single chunk of the game world are emitted. In our evaluation, where the game world consists of over 9000 chunks (96x96), more than 9000 LLIs need to be kept pending for synchronizing the world's game state. When considering that our evaluation scenario only covers a small fraction of a complete map in Minecraft ($\approx 3.6{\times}10^{15}$ chunks [7]) we see that direct update retrieval on chunk-level granularity for inter-server game state synchronization is not scaling well. Focusing on the traffic volumes in bytes, the reduced number of Interests in RMA reduces the traffic volume compared to the naive approach.

Nevertheless, the evaluation results for the NDN approaches show high overhead on packet size, mainly induced by Interests and by Data packet header fields. The low average Data packet size resulting from game state synchronization on a per chunk basis leads to a low payload size vs. header size ratio. In RMA the overhead could be reduced by decreasing the granularity of game state update packaging. Instead of sending updates for each world chunk in a single packet, updates for neighbouring chunks could be summarized (e.g., updates of 2x2 chunks per packet or multiples of that) in a single Data packet under a common name. Thereby, the number of required Interests would decrease while the payload size of Data packets would increase. Summarizing the game state updates of too many chunks, however, would require to split the summarized game state updates into multiple Data packets, leading again to increased overhead.

7 CONCLUSION AND FUTURE WORK

In this paper, we develop a system for inter-server game state synchronization using the NDN architecture. In our prototype, we use Minecraft as a real-world example of online games and utilize two NDN-based approaches for inter-server game state synchronization. Our evaluation shows that NDN beats IP-based implementations regarding the resulting traffic volume when utilizing NDN's inherent multicast functionality despite NDN's higher protocol overhead.

Leaving inter-server game state synchronization aside, the next step on our roadmap towards information-centric gaming is using NDN for client-server communication. Reorganizing the game data requested by clients to take advantage of the semantic richness of names may enable multicasting benefits also for client-server communication. Besides multicast benefits, NDN-based client-server communication with its host independence could yield latency reductions and redundancy support as well. When client requests are forwarded directly to the server simulating the corresponding region, no indirection via inter-server synchronization as in current systems would be required.

Source code resulting from this work is published as open-source software and available in an online repository [10].

REFERENCES

[1] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto, C. Fan, C. Papadopoulos, D. Pesavento, G. Grassi, G. Pau, H. Zhang, T. Song, H. Yuan, H. B. Abraham, P. Crowley, S. O. Amin, V. Lehman, and L. Wang. 2016. *NDN-0021: NFD Developer's Guide.* Technical Report. https://named-data.net/publications/techreports/ndn-0021-7-nfd-developer-guide/

[2] Entertainment Software Association. 2019 (accessed 2019-05-09). *U.S. Video Game Sales Reach Record-Breaking $43.4 Billion in 2018.* http://www.theesa.com/article/u-s-video-game-sales-reach-record-breaking-43-4-billion-2018/

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

[4] ZeroMQ Community. 2019 (accessed 2019-05-09). *ZeroMQ - Distributed Messaging.* http://zeromq.org/

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

[6] S. Floyd, V. Jacobson, C. G. Liu, S. McCanne, and L. Zhang. 1997. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking* 5, 6 (1997), 784–803.

[7] Fandom Games. 2019 (accessed 2019-05-09). *Official Minecraft Wiki – The ultimate resource for all things Minecraft.* https://minecraft.gamepedia.com/Minecraft_Wiki

[8] MCDevs. 2019 (accessed 2019-05-09). *Protocol - wiki.vg.* https://wiki.vg/Protocol

[9] Jim Meyering and Paul Eggert. 2018 (accessed 2019-07-26). *GNU Gzip.* https://www.gnu.org/software/gzip/

[10] Philipp Moll. 2019 (accessed 2019-07-30). *ACM ICN 19 Reproducibility.* https://github.com/phylib/ACM-ICN-19-Reproducibility

[11] P. Moll, M. Lux, S. Theuermann, and H. Hellwagner. 2018. 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).* Article 1, 6 pages.

[12] P. Moll, S. Theuermann, H. Hellwagner, and J. Burke. 2019. Distributing the Game State of Online Games: Towards an NDN Version of Minecraft. In *2019 IEEE International Conference on Communications Workshops (ICC Workshops).* 1–6.

[13] Named Data Networking. 2019 (accessed 2019-05-09). *Mini-NDN: A Mininet based NDN emulator.* https://github.com/named-data/mini-ndn

[14] PrismarineJS. 2019 (accessed 2019-05-09). *Create Minecraft bots with a powerful, stable, and high level JavaScript API.* https://github.com/PrismarineJS/mineflayer

[15] SpigotMC Pty. 2019 (accessed 2019-05-09). *Spigot MC – High Performance Minecraft.* https://www.spigotmc.org/

[16] W. Shang, A. Afanasyev, and L. Zhang. 2018. *NDN-0056: VectorSync: Distributed Dataset Synchronization over Named Data Networking.* Technical Report. https://named-data.net/publications/techreports/ndn-0056-1-vectorsync/

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

[18] A. Yahyavi and B. Kemme. 2013. Peer-to-Peer Architectures for Massively Multiplayer Online Games. *Comput. Surveys* 46, 1 (Oct. 2013), 1–51.

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

[20] M. Zhang, V. Lehman, and L. Wang. 2016. *PartialSync: Efficient Synchronization of a Partial Namespace in NDN.* Technical Report NDN-0039. NDN.

[21] Z. Zhu and A. Afanasyev. 2013. Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking. In *2013 21st IEEE International Conference on Network Protocols (ICNP).* 1–10.