Computer Networks 185 (2021) 107723

Contents lists available at ScienceDirect

Computer Networks

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

A Quadtree-based synchronization protocol for inter-server game state synchronization

Philipp Moll [a,*], Selina Isak [a], Hermann Hellwagner [a], Jeff Burke [b]

[a] Alpen-Adria-Universität Klagenfurt, Austria
[b] UCLA, CA, USA

ARTICLE INFO

Keywords:
Named Data Networking
Distributed dataset synchronization
Online games

ABSTRACT

Online games are a fundamental part of the entertainment industry but the current IP infrastructure does not satisfactorily fulfill the needs of these services. The novel networking architecture Named Data Networking (NDN) inherently supports network-level multicast and packet-level security and thereby introduces promising features for online games. In this paper, we propose an NDN-based approach to synchronize game state in a server cluster, a task necessary to allow multiple players in large numbers to play in the same game world. The proposed *Quadtree Synchronization Protocol* applies NDN's data-centric nature to decouple the game world from the game servers hosting it. This means that requesting changes of a specific game world region becomes possible without knowing which game server is responsible for the requested region. We use a hierarchic game world structure when requesting data that allows the network to forward requests to the responsible game server without directly addressing it. This region-based naming scheme decouples world regions from servers which eases the management of the game server cluster and allows easier recovery after server failures. In addition, this decoupling allows exchanging information about a geographical region, such as a game world, without knowledge of the other participants changing the world. Such a region-based synchronization mode is not possible to implement with existing protocols. However, it allows building distributed systems that do not require a central server to work. Besides architectural benefits, network emulations show that our protocol increases the efficiency of data transport by utilizing network-level multicast. Our proposed approach can keep up with current protocols which can be used for inter-server game state synchronization.

1. Introduction

Over the last decade, computer games have become a fundamental part of the entertainment industry and an everyday commodity to a large part of the population. In 2019, 65% of the adult American population – in other words, about 135 million Americans – played games [1]. Of those, 63% played games together with their friends, which shows the important role of online computer games. When focusing on technical aspects of computer games, however, we see that the networking part of modern games still relies on decades-old technologies, which were never intended to be used in games and are often part of the cause of overloaded and crashing game servers during peak hours.

In Massive Multiplayer Online Role-Playing Games (MMORPGs), thousands of players share an extremely large virtual world. Since a single server would not be capable of handling the computation load generated by the high number of players, the simulation of such game worlds is often distributed onto server clusters. This distribution requires the synchronization of relevant game state information among all servers in the cluster. When utilizing our current IP infrastructure for this task, all servers need to continuously send updated game state information to all other servers, resulting in a high amount of redundancy. Previous work [2] showed the potential for exploiting network-level multicast functionality, as provided by the novel networking architecture Named Data Networking (NDN) [3], to effectively reduce the total amount of network traffic for games. We suggest that utilizing NDN for game state synchronization could bring advantages reaching beyond reducing inefficiencies only. One of those advantages is the possibility to decouple game state information from the server producing it, which allows reducing the complexity of distributed game server architectures. The reduced complexity leads to easier recovery after server failures and the flexibility to dynamically

* Corresponding author.
E-mail addresses: philipp.moll@aau.at (P. Moll), selina.isak@aau.at (S. Isak), hermann.hellwagner@aau.at (H. Hellwagner), jburke@remap.ucla.edu (J. Burke).

https://doi.org/10.1016/j.comnet.2020.107723
Received 1 July 2020; Received in revised form 6 November 2020; Accepted 29 November 2020
Available online 1 December 2020

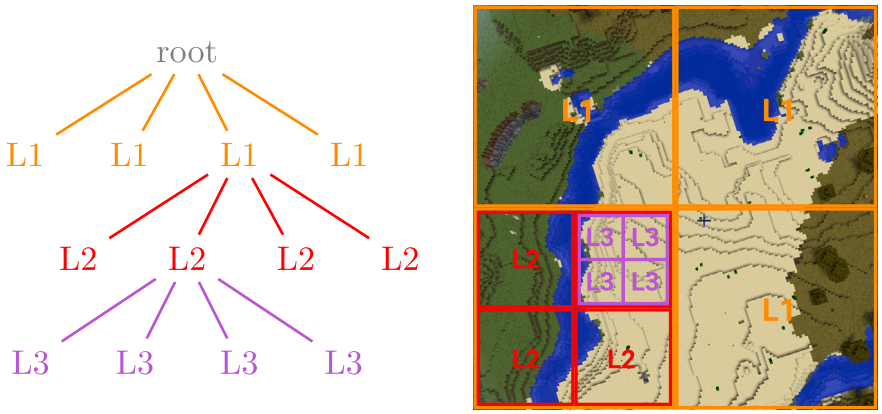


Fig. 1. Example quadtree, as used by QSP, to structure game worlds. With every additional tree level, the area covered by tree nodes becomes smaller.

add or remove servers from the cluster to overcome shortages in computational capacity.

We argue that one key concept required to build a distributed online gaming architecture is the synchronization of game state information in the server cluster, which we refer to as *inter-server game state synchronization* (GSS). Building on the insights gained in our previous work [4], we now propose a highly scalable information-centric system for inter-server game state synchronization called *Quadtree Synchronization Protocol* (QSP). We focus on achieving high scalability in terms of supporting large game worlds, as well as a high number of servers participating in the simulation of game worlds. We achieve this scalability by creating a fully decentralized system, which works without a synchronization master or any other kind of central component. This decentralization pays off because the system is not limited by the reliability, throughput, and computational power of a single central component.

Our proposed protocol QSP is an NDN-based synchronization protocol tailored to synchronize geographically structured data. One possible field of application, which is the focus of this paper, is game state synchronization, where QSP's advantages are twofold: (1) its architectural features simplify application design, and (2) it increases efficiency to reduce network load. One architectural feature is the use of application-level naming that allows sync participants to synchronize the content of geographic regions without knowing which other participants are responsible for those regions, or without knowing the other participants at all. For online games, this allows changing the game servers' region responsibilities during a game without impacting the game state synchronization process. This increased flexibility allows building more dynamic and fault-tolerant server clusters. Besides architectural advantages, QSP's design aims to increase efficiency by making use of NDN's inherent network-level multicast functionality. This reduces duplicated traffic and leaves more server bandwidth for other types of game-related communication.

The core of the synchronization protocol is a quadtree that structures the region to synchronize in a hierarchic way. While the tree's leaf nodes represent small areas of the overall region and contain all objects of those areas, nodes on higher levels of the tree stretch over multiple leaf nodes and thereby cover larger areas. Finally, the tree's root node covers the overall region to synchronize, as indicated in Fig. 1. In addition, the quadtree is built as a Merkle tree, meaning that every non-leaf node of the tree is assigned a cryptographic hash value, which is calculated by hashing the child nodes of the corresponding node. Thereby, a single changed object somewhere in the overall region is recognized by a change of the tree's root hash. Starting from the root, it is possible to track the change down to the single leaf node that changed.

One basic concept of QSP is that every participant in the synchronization process (e.g. every game server) picks a region of arbitrary size and publishes changes in that region. While this process suggests ownership of the region by the participant, a region's ownership is not essential for QSP to work since the synchronization process focuses on synchronizing the data of geographic regions and not on updates of individual participants.

While developing QSP, a focus on inter-server game state synchronization as a use case was set. However, we want to highlight that the protocol can be used for any other application having geographically structured data. Examples for such applications are geographic information systems (GIS) where large amounts of data are generated all over the world, or mirror worlds—AR/VR overlays covering the real world, where potentially everyone can contribute or consume data.

In this paper, we introduce relevant background information and technologies, such as NDN, as well as state-of-the-art synchronization protocols as related work in Section 2. In Section 3, QSP is described in detail. Subsequently, in Section 4, we compare the proposed protocol to other state-of-the-art protocols. We argue that the real benefits of QSP lie beyond performance measures only and cannot be assessed by quantitative comparisons. This is why we showcase the capabilities and flexibility of our protocol in a second scenario, where game servers have no knowledge about the other servers of the synchronization process, in Section 5. Finally, in Section 6, we recapitulate our findings and discuss potential future directions.

2. Background

When developed in the 1960s, the primary use for the Internet – or, to be more precise, of the TCP/IP protocol suite – was to enable point-to-point communication between endpoints. Today, data distribution tasks, such as video delivery in multimedia streaming applications, supersede point-to-point communications. Although the way we use the internet has changed, no major updates to its architecture have been implemented since its invention, rendering it as poorly suited for nowadays use. One missing link that would increase efficiency for many use cases is reliable network-level multicast. In online games especially, in which a central server sends data about the current game state to all players, network-level multicast has a high potential [2].

2.1. IP-based game server cluster solutions

The synchronization of game server clusters is one use case that would largely benefit from network-level multicast, as described by Cronin et al. [5]. Using a client/server architecture with a single game server introduces a potential bottleneck and a single point of failure. To avoid these issues, the single game server is often replaced by a server cluster. Fig. 2 depicts a typical game server cluster in which game state management is distributed to multiple servers. Every server is responsible for managing a certain region of the world. Clients then

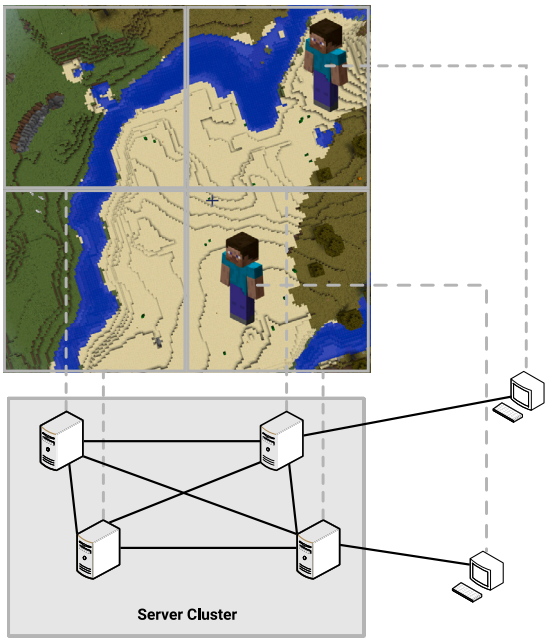


Fig. 2. An online game server cluster, in which the individual regions of the game world are hosted by different servers.

connect to the server which is responsible for the region hosting their avatar. In such game server clusters, it is essential that every server knows the current state of the entire, overarching game world. Thus, synchronization of game-relevant data between all servers in the cluster is required. For this inter-server game state synchronization, network-level multicast would be beneficial. However, this is only possible on a limited scale in today's IP infrastructure.

Several alternatives to classical client/server architectures for games have been researched. One promising alternative may be spatial publish –subscribe systems, like discussed by Hu et al. [6]. In the S-VON system, a virtual world is managed by super peers handling a spatial publish–subscribe system. Clients can subscribe to events happening in regions in the virtual world and publish events at specific locations. Whenever a client generates an event, it is sent to a super peer, which then forwards it to all clients subscribed to the region where the event occurred. One advantage of S-VON is the possibility to use multicast when distributing events. This is implemented by super peers forwarding events to the clients via other super peers in IP networks. This procedure, however, requires the management of large subscription tables. With QSP, we follow the idea of a spatial publish–subscribe system and implement it without a need for super peers or subscription tables.

2.2. Towards gaming over NDN

Reliable network-level multicast, application-level naming, and packet-level security are only three among many features provided by the novel information-centric network architecture called Named Data Networking (NDN) [3]. In NDN, every data item is assigned a system-wide unique *name*. Instead of requesting data from a specific host, the data item's name is used to request it. In the strict pull-based system, consumers issue *interest* packets to request data items with a given name. Those interests are forwarded through the network on a hop-by-hop basis until they reach a node that has a copy of the requested data item. The data item is then sent back to the requester as a *data* packet on the reverse path of the interest. This is possible because interests leave traces on every forwarding node, which are realized as entries in the so-called *pending interest table* (PIT). Network-level multicast becomes possible by aggregating interests for the same name in the PIT. When a forwarding node receives an interest, the incoming interface is stored together with the name in the PIT. When the corresponding data arrives, the PIT contains all interfaces where interests came from and the data is sent to each interface. Moreover, incoming data packets are cached by every node in the network, allowing that interests can be answered directly from a cache in the network, rather than by the original data producer.

This caching behavior allows to review the concept of application-level naming used by NDN. In connection-oriented architectures, a connection for information exchange between exactly two participants – the producer and the consumer in NDN vocabulary – is established. The connection can be used to exchange any kind of data between producer and consumer and stays active as long as information is exchanged. In NDN by contrast, a focus on the data itself is set and the data items' names are requested instead of the endpoints addressed. The data names semantically describe the data items based on the application context rather than indicate the producer or its location in the network. When requesting a data item, it does not matter whether the data is produced locally or on a server in a remote location. The network forwards the request based on the application-level name to the data's origin or to a cached copy. Decoupling the application-level name and the data's producer eases application design. When requesting multiple data items in a distributed application, for instance, NDN can forward every request to a different producer if required.

Focusing on security, NDN secures the data items themselves by adding a cryptographic signature binding the data item to its producer and optionally by encryption. Once data items are published, they can serve requests of multiple consumers. This also means that actions for establishing integrity or confidentiality are performed once per data item and do not need to be repeated for every consumer, as is the case in connection-oriented systems. These packet-level security measures, furthermore, allow to place the secured data in network-intern caches and to securely retrieve data from insecure sources (e.g. caches) without losing integrity or confidentiality.

In [7], we outlined an information-centric version of Minecraft,[1] which represents the basis for this work. Minecraft is a 3D sandbox block-building-game with online multiplayer capability. Fig. 3 shows a typical situation in the Minecraft game. Here, the player is interacting with non-player characters. As visible in the screenshot, the game world is built of cubic blocks representing different materials with unique properties. The world is procedurally generated and almost infinite in size. One feature that makes Minecraft interesting is that players have the means to change every single part of the game world. In the envisioned gaming architecture, the simulation of Minecraft's game world is distributed to a server cluster in a way that every server of the cluster exclusively simulates one region of the game world.

The role of NDN in this architecture is twofold: Firstly, NDN is used for the communication between servers in the server cluster. Every server simulates objects like player avatars, monsters, or growing trees in a certain region of the game world, which we call *the server's region*. For providing a smooth playing experience, the server requires knowledge about objects in the other servers' regions, referred to as *remote regions*, for the simulation. We refer to the exchange of such game state information among the servers of the cluster as inter-server game state synchronization (GSS).

Secondly, NDN is for client–server communication. Since changes in the game world are the same for every Minecraft client, NDN's multicast functionality results in reduced traffic volume. Furthermore, when utilizing a server-independent naming scheme, fault tolerance increases since clients can request data depending on where they are in the game world, rather than from a specific server. A crashing game server thereby does not automatically result in connectivity issues. Despite the promising expectations for client–server communication, we focus on GSS in this paper.

[1] https://www.minecraft.net/, last accessed: 2020-05-22.

Fig. 3. A typical situation in the Minecraft game. The screenshot shows the world's block-based structure (best visible on terrain edges) and chicken as peaceful non-player characters.

2.3. Inter-server game state synchronization using NDN

In current systems, GSS relies on TCP channels between all individual servers. Establishing such channels in NDN is neither possible nor meaningful. However, since multiparty communication is an intrinsic feature of NDN, a multiparty equivalent to TCP channels for data distribution exists. Distributed dataset synchronization protocols (referred to as *sync protocols*) are one possibility to distribute data reliably to multiple consumers. In the following, we look at existing sync protocols, which might support GSS in NDN.[2]

ChronoSync [8] was the first NDN-based sync protocol. ChronoSync keeps track of the dataset's state by using a digest tree. Every participant of the sync process contributes data and is responsible for managing an increasing version number, representing the state of its data. Whenever a participant changes its data, the corresponding version number increases and changes the digest of ChronoSync's digest tree, whereby all participants are notified of changes in the dataset.

VectorSync's [9] design improves on weaknesses of its predecessor ChronoSync. In VectorSync, every data item of the distributed dataset is assigned a version number and is referred to as a data stream. A state vector, maintained by every sync participant, keeps track of all data stream versions and is periodically published via heartbeat messages.

StateVectorSync (SVS) [10] improves on VectorSync and focuses on supporting disruptive networks. Similar to VectorSync, state information of every data stream is encoded in a state vector. By eliminating VectorSync's need for a group leader, SVS becomes capable of functioning in disruptive networks, found, for instance, in disaster recovery scenarios.

PartialSync (PSync) [11] follows a slightly different concept than the sync protocols described above. In PSync, a single producer keeps track of multiple data streams, and consumers can either subscribe to all data streams or a subset only. By using invertible bloom filters for encoding the dataset's state, PSync becomes scalable in terms of supporting high numbers of consumers as well as data streams. Its scalability and practicability for a wide range of applications is also demonstrated by its use in the Named-data Link State Routing protocol (NLSR) [12].

For GSS, decoupling a region from the responsible server is important to provide flexibility in terms of changing region responsibilities. For instance, a server requesting changes from the region next to its own region should be able to request the region's data directly, without the need to know which server is managing it. This way, the server can continue requesting changes from the neighboring region, even if the owner of the region changes. In the presented sync protocols, a server is responsible for managing a single data stream that contains all data managed by the server. This binds all objects in the server's region to the data stream and directly couples the region to the server.

In previous work, we introduced the **Region Manifest Approach** (RMA) [4] as the first sync protocol intended for GSS and tailored for the geographically structured game state information found in games. The RMA approach decouples the game world from servers by establishing sync regions. Sync regions can be directly addressed by their hierarchic names, instead of a detour over the server responsible for the region. Working with RMA showed that further improvements towards supporting larger game worlds are required. More details on RMA and ways to improve it are given in Section 3.

3. Quadtree-based synchronization

In this section, we discuss the design of our proposed *Quadtree Synchronization Protocol* (QSP) that allows building game server clusters at scale. QSP is based on the quadtree data structure and aims to provide scalability with respect to supporting the synchronization of large game worlds and a high number of participants in the sync process. This is achieved by using insights from previous work and improving it, as explained in the following.

3.1. Lessons learned from RMA

When following the Region Manifest Approach (RMA) [4], the game world is divided into so-called map chunks. Multiple of those disjointed small-scale areas are summarized in server regions, each managed by exactly one server. For synchronization, every map chunk is handled as an individual data stream and its version number is increased whenever an object in the map chunk changes. Although map chunks can be handled as individual data streams, the number of map chunks becomes excessive in large game worlds. Most current sync protocols are not capable of handling such high numbers of data streams.

The RMA exploits the geographic relation between individual map chunks (e.g. neighborhood) for more efficient synchronization. Multiple map chunks are grouped into sync regions and every sync participant is assigned the ownership of multiple sync regions. Changes in those larger sync regions are summarized as individual data streams and communicated to all other sync participants. However, the RMA is forced to a trade-off between sync region size and the total number of sync regions. If the sync region size is large, the amount of changes in sync regions increases and becomes problematic, since huge sync data packets are generated. In addition, large sync regions reduce the flexibility of dynamically resizing server regions because a single sync region cannot be split onto multiple servers. In contrast, decreasing sync region sizes affects the total number of sync regions, and synchronizing too many of them simultaneously does not scale well.

Learning from RMA, we see that sync regions need to be variable in size. To support decoupling the game server from its region, the name representation of a region needs to include the region's position in the game world. Moreover, it must not contain a reference to the server responsible for the region. One possibility to structure geographic regions found in many GIS applications are R-Trees [13]. They allow fast querying of nearby objects in a region, but an R-Tree's structure is formed by the existing data and thereby does not allow inferring non-ambiguous region names. Quadtrees – a tree data structure in which every node has exactly four children – partition a two-dimensional space hierarchically and allow fulfilling both requirements: creating sync regions of variable size and inferring unique names for those regions. This makes quadtrees a perfect candidate for GSS. In the next section, details on how to make use of quadtrees for GSS are provided.

[2] Please note that the terms server and sync participant are used interchangeably.

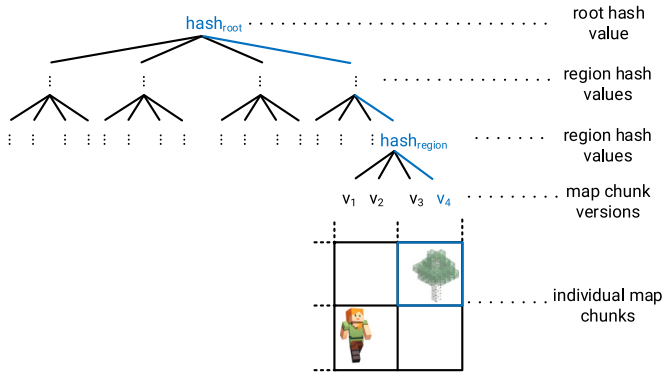


Fig. 4. Quadtree for structuring the game world. Individual map chunk versions are stored in the tree's leaf nodes. Intermediate nodes are assigned hash values allowing to track changes in the tree.

3.2. Basic protocol function

Game worlds can often be seen as two-dimensional spaces. When structuring a game world with a quadtree, the quadtree's root covers the whole two-dimensional space, as indicated in Fig. 1. Each of the root's children covers a quadrant of the root's space. This division of the space is continued recursively until the highest degree of granularity is reached. In QSP, sync participants can be assigned any region, represented by a quadtree node. Map chunks, representing the smallest unit to synchronize, are the leaf nodes of the quadtree and are assigned version numbers. Whenever a map chunk changes, the map chunk's version number is increased. The central role of QSP is to communicate the information about the increased map chunk version to all sync participants. Therefore, the quadtree is built as a Merkle tree [14], where every non-leaf node is assigned a cryptographic hash of all children. Fig. 4 visualizes the structure of the used quadtree. The individual map chunks contain the game state of the corresponding area and their versions are stored as leaf nodes in the quadtree. When the game state of a map chunk changes (e.g. an avatar is moving), the map chunk's version increases and causes all hash values on the path from the map chunk to the root to change.

QSP is utilizing the possibility to track down changes by observing hash values of tree nodes. Every sync participant is managing a copy of the quadtree covering the whole game world. A sync participant's region is represented by a quadtree node, meaning that the sync participant is only changing map chunks in the subtree starting with the corresponding node. In contrast, the sync participant is interested in receiving changes of all subtrees except its own. In Fig. 5, a sync participant is responsible for the left upper quadrant of the game world and is interested in the other quadrants of the quadtree. Initially, no knowledge about the remote regions is available and the sync participant assumes that all map chunks of the remote regions are in their initial state. By requesting the hash values of the remote regions and by comparing the received hash values to those of the corresponding node in the local quadtree, the sync participant can decide whether a region has changed or not. If a remote region has changed, a recursive process of requesting hash values from lower-tier tree nodes (smaller regions) is employed, until the lowest level is reached, and the exact location of the changed map chunk is identified.

3.3. Naming regions of the map

Requesting hash values from remote regions becomes possible by utilizing a simple naming scheme that incorporates the hierarchy of the quadtree. This allows us to address every tree node and thereby regions of varying size. The root node of the quadtree is identified by the application prefix (e.g. `/prefix/`). When traversing the tree,

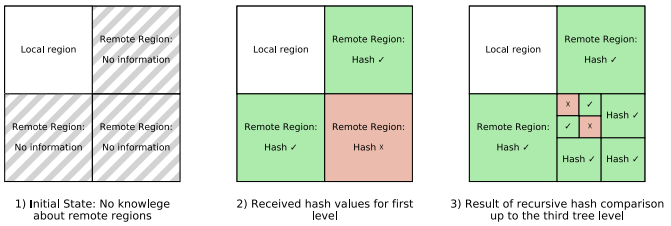


Fig. 5. Tracking changed regions by comparing hash values on different levels of the quadtree.

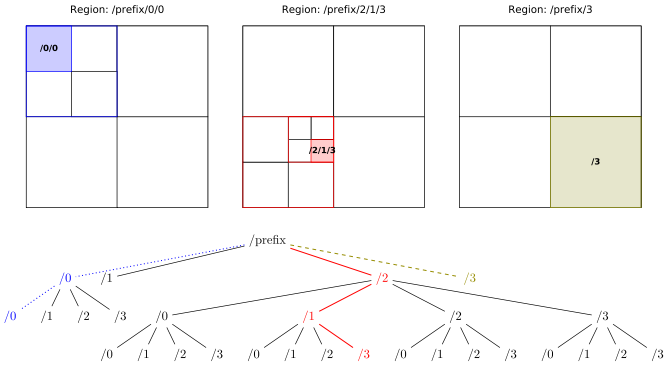


Fig. 6. The naming scheme following the hierarchical tree structure allows naming sync regions of varying size.

the index of the corresponding child node, ranging from 0 to 3, is appended on every tree level. In an NDN name identifying a region, every tree level is represented by a name component and stored by a single character only. Examples of such names and the relation between quadtree nodes and map regions are illustrated in Fig. 6. As shown in the image, the more tree levels are encoded in the name, the smaller the corresponding map region.

3.4. Detailed protocol design

After outlining the basics of using a quadtree for detecting changed regions, we continue with a discussing of QSP's primitives. Since NDN is a strictly pull-based system in which every data item needs to be requested by an interest, data cannot be pushed through the network without a prior request. While this feature fosters flow control and can be used to react to congestion, designing protocols for inherently push-based services becomes more challenging.

When breaking up GSS with quadtrees into smaller pieces, we see that two separate types of information are required. First, region hash values are required for comparison in order to decide which remote regions changed. Second, a sync participant needs to know which map chunks of the region has changed and receive information about their newest versions when a changed region is identified. Hence, the versions of changed map chunks need to be transferred as well. These two tasks are reflected by QSP and described in the following.

During the sync process, a sync participant requests changes from remote regions by issuing interests. These interests, referred to as *sync requests*, carry two parts of information: the name of the requested region and the requester's hash of that region, whereby the hash can be used to infer the region's state on the requester's side. Depending on the requester's state, the region owner receiving the sync request decides whether the *sync response* contains either map chunk versions (*chunk data response* (CDR)) or hash values of lower-level regions (*subtree hash response* (SHR)). The structure of sync requests and sync responses is shown in Fig. 7. Sync requests are implemented as NDN interests, and the information carried is encoded in the interest's name. A sync

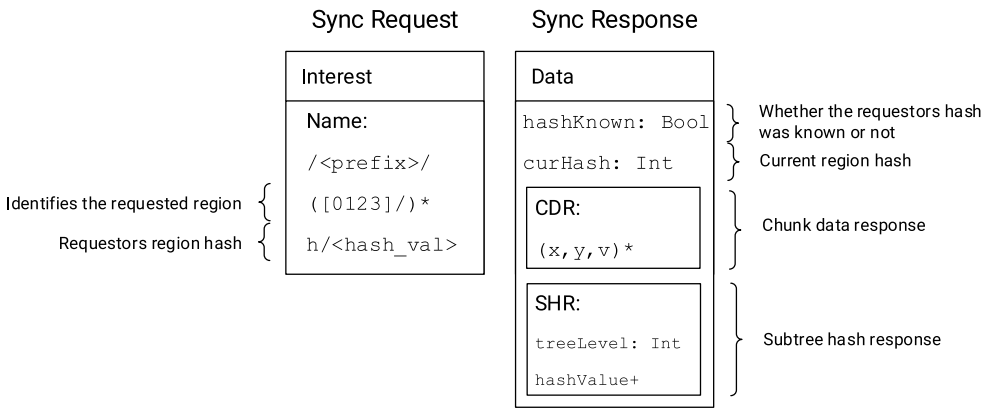


Fig. 7. Structure of sync requests and sync responses. A sync response contains either map chunk data or subtree hash values.

Algorithm 1: Handling a sync request

```
Data: syncRequest, currentQuadtree
Result: syncResponse
1  requestedRegion = currentQuadtree.get(syncRequest.region);
2  if syncRequest.hash = requestedRegion.hash then
       /* No change occurred                                  */
3      return null;
4  if syncRequest.hash = requestedRegion.outdatedHash then
       /* Found existing outdated hash                        */
5      changes = requestedRegion.storedChanges();
6      if size(changes) ≤ SHR_THRESHOLD then
7          return changes;
8      else
9          return requestedRegion.changedSubtreeHashes();
10 else
       /* Requester hash unkown                               */
11     return requestedRegion.subtreeHashes();
```

response is carried in the payload of an NDN data packet, whereby a response can either contain a CDR or an SHR.

Algorithm 1 sketches how region owners react to sync requests. By comparing the region's current hash to the requester's hash, the region owner can infer the requester's state and decide on further procedure. If the requester's hash is the same as the region's current hash, the region has not changed and no sync response is necessary (L2–L3). If the hash values do not match, at least one map chunk in the requested region has changed. For locating changes faster, a region owner recalls outdated hash values and keeps track of map chunk changes that occurred since those hash values. If the requester's hash value matches an outdated hash, the region owner can reply the changed map chunks (L4–L5). Since the requested region can be large in size and might contain a large number of changes, the region owner can decide whether to directly reply with the changed map chunk versions (a CDR is generated) (L6–L7) or to defer the delivery of changes to lower tree levels. If the number of changes lies above a certain threshold, the region owner responds with an SHR only (L8–L9). If the requester's hash is unknown, the region owner cannot infer changes and responds with hash values of lower tree levels (L10–L11). With these hash values, the requester can issue subsequent sync requests for lower-level subtrees (smaller regions in the originally requested region) and thereby track down changed parts.

For a region owner, storing outdated hash values together with changed map chunks is expensive. Hash values and changes need to be stored on every tree level and updated whenever a hash value changes. In a large quadtree, the space required for remembering past changes would rapidly exceed several gigabytes of memory. This is why QSP only stores a single outdated hash in every tree node. Our evaluations in Section 4 show that such a short history is sufficient for GSS.

3.5. Determination of QSP's parameters

Focusing on the above protocol, two parameters for tuning the sync process with quadtrees can be identified: (i) the threshold for deciding whether CDRs or SHRs are replied (referred to as *SHR threshold*). (ii) the tree level difference between sync request and subtree hashes in SHRs (referred to as *SHR level difference*), as described in the following.

3.5.1. Parameters of QSP

The specification of both parameters, visualized in Fig. 8, influences how many consecutive requests are required to identify and retrieve changed map chunk information from remote regions. When a region owner receives a sync request, the region owner can already identify which map chunks changed and could reply with a single CDR containing all changes. In this case, the sync process is finished after a single response because the requester receives all the required information in a single packet. The response, however, could be large in size and might exceed the MTU. The SHR threshold is used to restrict the number of changes per response and to defer the transport of chunk data to sync requests for lower tree levels. A high SHR threshold increases the response size, a low SHR threshold on the other hand leads to a high number of successively required sync requests. This is why we argue that the SHR threshold should be as high as possible, but small enough to keep responses to a reasonable size.

As stated above, an SHR contains hash values of lower-level subtrees. Intuitively, the hash values of the direct child nodes of a region could be enumerated in SHRs. In a quadtree, every region has only four children. This means that small packets including only four hash values would be sent when replying with SHRs when focusing on direct children only. A quadtree covering a large game world has many levels. Traversing such a tree when progressing one level per sync request would require many consecutive sync requests. This is why an SHR does not necessarily contain the direct children of a region, but hash values of several levels lower in the tree. The level difference between the requested region and the nodes of which hash values are returned is referred to as the SHR level difference. While a low level difference leads to a high number of SHR responses for traversing the quadtree, a high level difference could mean traversing the tree too fast which would increase the number of required CDR responses, each carrying only a small amount of changes.

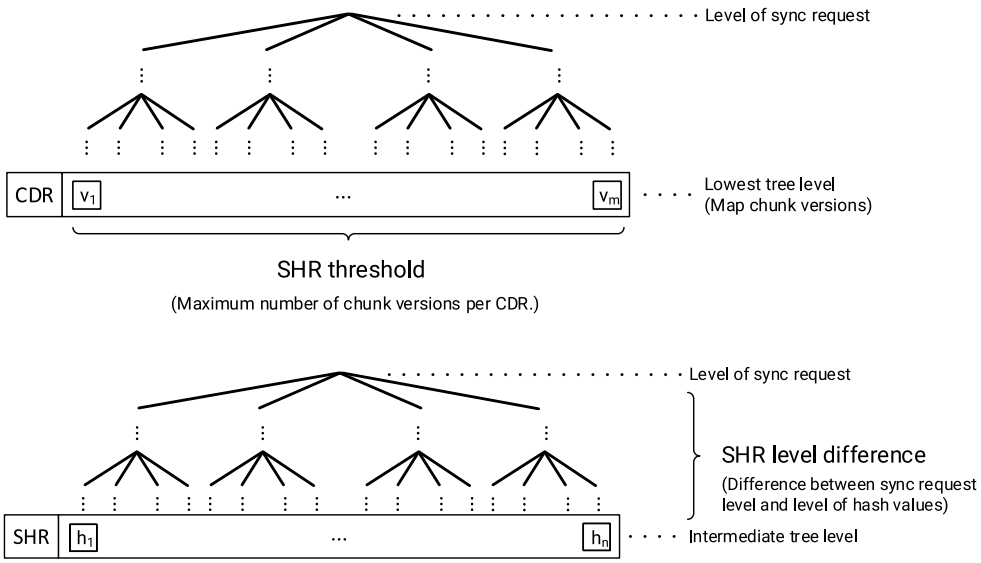


Fig. 8. Visualization of the parameters SHR threshold and SHR level difference.

3.5.2. Parameter study

For finding the best combination of these parameters, we conducted a parameter study. The goal of the study was to find values for all relevant parameters to reduce the total amount of required sync requests. For the study, an offline C++ prototype of QSP was created, where a Minecraft game world consisting of $2^{16} \times 2^{16}$ map chunks was mimicked. In the prototype, two copies of a quadtree spanning over the game world were managed. Game state changes, recorded on a real Minecraft server, were applied to one of the quadtrees and synchronized to the second quadtree using the above-presented protocol primitives. To be able to infer realistic packet sizes in the offline prototype, sync responses were encoded using protocol buffers[3] and compressed using zlib.[4] The parameter study showed that an SHR threshold of 200 changes per response leads to a maximum payload size of about 1200 bytes. Depending on the value of NDN's other data packet fields (e.g. the name), this payload should fit into an Ethernet frame. Thus, an SHR threshold of 200 appears to provide satisfactory results. Going above this value leads to packet fragmentation in typical networks, while lower values increase the total number of required sync requests

Interestingly, a high SHR level difference does not necessarily decrease the number of required sync requests. The size of the mimicked game world leads to a quadtree with 16 levels. When using an SHR level difference of 1, a maximum of 15 iterations of sync requests and SHR responses are necessary to retrieve changed map chunks. Considering the SHR threshold of 200 and the fact that not the entire game world changes at the same time, the number of required iterations is reduced. In the parameter study, the best results were achieved with an SHR level difference of 2, meaning that the sync response contains hash values of the tree nodes two levels below the requested region. Thereby the number of changes in consecutive sync requests is reduced fast enough so they do not exceed the SHR threshold and CDR replies can be sent. A higher value for the SHR level difference leads to consecutive sync requests on levels too low to be of interest so that only a small number of changes can be included in CDR responses, which subsequently requires more sync requests in total.

Based on the insights from the parameter study, evaluations presented in later sections are performed with an SHR threshold of 200 changes per update and an SHR level difference of 2.

[3] https://developers.google.com/protocol-buffers, last accessed: 2020-11-04.

[4] https://www.zlib.net/, last accessed: 2020-11-04.

3.6. Supporting forwarding scalability

One question we anticipate about our approach concerns forwarding scalability. In QSP, multiple sync participants collaboratively manage a geographic area, and the regions' names managed by the individual sync participants might be very similar. For instance, when two sync participants manage extremely small bordering regions, the two regions' names share a long prefix and only differ in the last name components. This similarity between names that are managed by different nodes can lead to challenges when forwarding sync requests. On NDN nodes, the longest prefix match algorithm is used to compare the name of incoming interests to the node's *forwarding information base* (FIB)—a data structure containing information about which data is likely to be found via which interface. To correctly forward a sync request for one of the small-scale bordering regions to its sync participant, the name uniquely identifying the region has to be in the FIB. This means that the FIBs of all network nodes connecting the sync participants require FIB entries for every region managed by sync participants. While this might not be an issue when sync participants are located in a data center, a distribution of the sync participants over a continent might lead to the FIBs of backbone nodes growing to a challenging size.

In [15], it is argued that forwarding scalability issues might not arise in NDN. *Forwarding hints* [16] are one possibility to support forwarding on backbone nodes. When using forwarding hints, a differentiation is made between global routable names (locators) and the actual data names (identifiers). Backbone nodes only handle locators and thereby the FIBs of those nodes stay reasonably small. When a consumer requests data, a locator is added to the emitted interest as a forwarding hint, telling the forwarding nodes where the data might be located. The interest is then forwarded to the location indicated by the hint. From the node representing the locator on, the locator is removed from the interest and the actual name is used for forwarding. However, when requesting data with this approach, a mapping between data names and locators is required. Therefore, special lookup services, such as NDNS [17], can be used, or the mapping can be implemented on the application level.

An alternative to forwarding hints is self-learning, as suggested in [15]. When using self-learning, interests are forwarded through the network until no FIB entry for the corresponding interest is found on a node. This node then floods the interest to random interfaces to eventually reach the producer. Returning data packets indicate a way to the data and allow nodes to add new FIB entries.

For QSP, both discussed approaches are usable, whereby forwarding hints appear favorable since they allow more control for sync participants. However, a combination of both approaches might be possible. For instance, forwarding hints could be used when locators are available and to handle fast churn (e.g. due to changes of region responsibilities), the locator can be omitted in order to let the network figure out the correct sync participant's location.

4. QSP in a game server cluster

Server clusters are the classical setup for online games. In these cases, the simulation of the game's progress is distributed to multiple servers of a cluster and the computational load is thereby shared among multiple server instances. The servers of the cluster are connected via high-bandwidth connections so that the game's distributed architecture is transparent for the end-user playing the game. In this section, we demonstrate our prototype QSP implementation for GSS in a server cluster, where the individual servers are handled as QSP sync participants. Although the real advantages of QSP are of architectural nature and not measurable in performance comparisons, we compare our implementation to an IP-based and to an NDN-based alternative. This shows that QSP's performance is comparable to state-of-the-art protocols.

4.1. The emulated server cluster setup

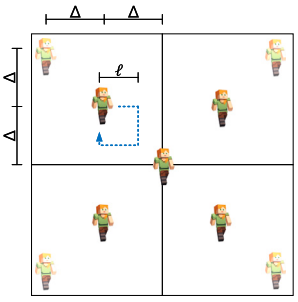
To evaluate the performance of QSP, we create an emulated game server cluster using the Mini-NDN network emulator.[5] A Minecraft game world consisting of $2^{16} \times 2^{16}$ map chunks[6] is collaboratively simulated by all sync participants in the server cluster, where every sync participant is responsible for one region of the game world. In our evaluation, two types of server cluster configurations are considered. In the first configuration—*the data center setup*—all sync participants are connected via a central node in a star topology. In the second configuration—*the continental setup*—sync participants are randomly distributed over a continent in the GÉANT research topology.[7]

To facilitate repeatability, no real Minecraft server and Minecraft clients are used in the evaluation. Instead, the behavior of a real Minecraft game world is first observed and the game state changes in this world are recorded. In our emulation, every map chunk is assigned a version number and GSS is implemented by synchronizing the versions of all map chunks across the server cluster.

In the evaluated game server cluster, three different synchronization approaches are compared. In the QSP approach, a quadtree spans over the entire area of $2^{16} \times 2^{16}$ map chunks. The SHR threshold and SHR level difference parameters are set as described in the parameter study in Section 3.5. Sync participants send sync requests for every remote region and consequently to lower-level regions until the game world is in sync, as described in Section 3.

The second approach we evaluate is the NDN sync protocol State-VectorSync (SVS). The protocol keeps track of version numbers for every data stream in the distributed dataset. To decouple map chunks from sync participants, it would be necessary to handle every single map chunk as a data stream. However, SVS uses a state vector containing the current version number of every data stream. Focusing on a game world with $2^{16} \times 2^{16}$ map chunks, the resulting state vector would be too large to be processed on a server or be sent over the network.[8] Therefore, we bind map chunks to the sync participants' namespaces in our SVS implementation. Therefore, a sync participant summarizes all map chunks in its region as a single data stream. Whenever a map chunk's version is increased, the sync participant increases its data stream's version and publishes the new version via SVS. All information about the individual map chunks that changed in the sync participant's region is published as a separate data packet and is retrieved via interest packets. The corresponding data packets carrying the updated map chunk versions are encoded with protocol buffers and compressed using zlib—the same way as CDRs are encoded in QSP. For evaluation, a slightly modified version of the C++ SVS prototype implementation[9] is used.

For the IP implementation, the ZeroMQ networking library (ZMQ)[10] is used in pub/sub mode. In our Python-based implementation, every sync participant pushes updates in its region as soon as they happen. In addition, sync participants subscribe to the updates of all other sync participants in the server cluster.



Scenario	Δ	ℓ
concentrated	20	50
widespread	5,000	50
max distance	50,000	50

Fig. 9. Visualization of client movements and parameters for generating change traces.

4.2. Tracing Minecraft's game state

Although we do not use a real online game, we want to mimic changes in an emulated game world as closely as possible to changes happening in such a game world. Therefore, we observe how a Minecraft game world changes during a real play session. In Minecraft, players, via their avatars, cause objects in the game world and subsequently the game world itself to change. By default, only those parts of the game world where player avatars are located are simulated. All other areas remain static; their game state does not change. Areas that are actively simulated are lively: Besides the player's avatars, other objects change frequently too. Examples for such objects are growing trees or monsters roaming around freely.

To test the performance of QSP under different conditions, we observe Minecraft's game state in three scenarios that represent different playstyles found in online games. The first scenario—the *concentrated scenario*—represents a collaborative online game where multiple players interact in the same area. This is commonly seen when players collaboratively solve tasks. In the *widespread scenario*, players act individually in relatively close regions of the game world. This playstyle is often seen in games, such as Minecraft, where players share the same world while their avatars do not necessarily directly interact with each other. In the *max distance scenario*, players inhabit the same game world, but their avatars are very distant, so that they cannot interact with each other.

To create these scenarios, we use emulated Minecraft clients that behave in a predefined manner. The emulated clients are based on the

[5] Mini-NDN is a Mininet-based emulator specialized for experimentation with the NDN architecture. https://github.com/named-data/mini-ndn, last accessed: 2020-10-20.

[6] A map chunk in Minecraft is an area of the game world with the size of 16 × 16 meters. Map chunks are an important data structure for Minecraft's server application and are used, for example, when keeping parts of the game world persistent.

[7] https://www.geant.org/Networks/Pan-European_network/Pages/GEANT_topology_map.aspx, last accessed: 2020-10-20.

[8] The memory demand for unsigned integer version numbers of 2^{32} data streams (4 bytes each) sums up to more than 17 GB.

[9] https://github.com/jonnykong/Sync-MANET, last accessed: 2020-10-20.

[10] https://zeromq.org/, last accessed: 2020-10-20.

mineflayer client emulator[11] and are programmed in a way that they join the game world at certain coordinates and move around in the shape of a square. As visualized in Fig. 9, the start coordinates of the clients are distributed in the shape of a cross and in all scenarios we emulate 40 clients walking around for about ten minutes. The start coordinates of the clients differ between scenarios. In the concentrated scenario, Δ is set to 20 m, which means that clients can see the other avatars. In the widespread scenario, Δ is increased so that clients cannot see the other avatars but are still relatively close in the game world. In the max distance scenario, Δ is set so that the avatars are distributed on a $2^{16} \times 2^{16}$ map chunk large section of Minecraft's game world. Finally, by using a Minecraft plugin for tracing changes in Minecraft's game world originating from previous work [4], a ten minute change trace for each of the three scenarios is created. To create the change traces, a snapshot of the game world is created every 500 milliseconds and compared to the previous snapshot. Thereby, objects that have gone through changes can be identified and changes in the emulated game world can be retraced.

4.3. QSP's performance in the server cluster

The evaluation results indicate that each of the compared protocols has its own strengths and weaknesses. In Fig. 10, the accumulated payload of outgoing sync packets on all sync participants' nodes is visualized. For the NDN-based sync protocols, this means that the actual map chunk versions, but also payload from other sync protocol data packets (e.g. content of SHR responses) are counted. In the IP-based solution, no other payload than map chunk versions exists, leading to a slight inherent advantage of the ZMQ approach.

In the top most row of Fig. 10, the simulation of the game world is distributed to four sync participants, each managing a quadrant of the game world. For the two NDN approaches, we see that the accumulated payload sent from the sync participants' nodes is lower in the data center setup compared to scenarios where servers are distributed in the continental setup. This is a result of NDN's inherent multicast functionality. While all traffic flows over the central node in the data center topology and favors multicast, fewer shared links exist in the European research topology. This reduces the possibilities for multicast benefits and leads to a higher traffic volume.

Comparing QSP to SVS in the four-server-scenario, we see that less payload is generated by SVS. While SVS's advantage is only negligible in the concentrated scenario, the advantage becomes larger in scenarios where client avatars are distributed over wider areas of the game world. In the concentrated scenario, only a small part of the game world, respectively of the quadtree in QSP, underlies changes. Since only hash values of changed subtrees are included in QSP's SHR responses, fewer sync requests, and thus less payload is generated in the concentrated scenario. In SVS, no geographic relations are considered, which is why the client distribution does not affect SVS performance.

Comparing the NDN-based approaches to ZMQ, we see that the sent payload is reduced largely by network-level multicast in the data center setup, especially when focusing on SVS. However, the potential reduction of payload is not as large as in the SVS approach due to the high protocol overhead of QSP. In the continental setup where multicast benefits are smaller, both NDN approaches produce about the same amount of data as our IP-based solution. In the continental setup, the potential multicast benefits also depend on the sync participants' location in the network that is varied in consecutive evaluation runs. This explains the higher variability of results for the NDN approaches in these evaluations.

The bottom row of Fig. 10 shows the results of a server cluster consisting of 16 servers, where all servers manage a region of equal size. Unfortunately, SVS performs poorly in this setup, most likely resulting from the high amount of changes during short intervals. SVS was primarily designed for dataset synchronization in disruptive networks and benchmarked with fairly low data generation rates. In the 16-server-scenario, data is simultaneously generated on multiple servers, resulting in SVS not working as expected. On top of the increased traffic volume of SVS, a high number of updates is lost, resulting in incomplete synchronization.

QSP, however, performs well in the 16-server-setups and multicast gains become stronger. While every update needs to be sent to every other server in the IP implementation, a high amount of packets profit from network-level multicast, reducing the sent payload when using QSP.

Fig. 11 visualizes the number of incoming interest and outgoing data packets and thereby allows to shed light on NDN's protocol overhead. When comparing the amount of sent packets between QSP and SVS, we see advantages of SVS in the four-server-setting. The reason for this is hidden in the protocol details: In SVS, every sync request and every sync response carry a state vector including the state of all participants. Thereby, information about changes in the entire game world can be inferred from every sync packet. In QSP, however, sync requests and sync responses only carry information about a certain region. Information about other regions must be requested separately. Another reason for the increased number of packets in QSP is that regions with many changed map chunks require several SHR iterations before map chunk versions are transmitted. However, QSP's overhead pays off in the 16-server-setup. Here, QSP's initial sync requests target smaller regions than in the four-server-setup. This leads to fewer required SHR responses and thereby the total number of required sync requests is reduced leading to good results of QSP. SVS, on the other hand, fails to synchronize changed map chunks in the 16-server-setup, as described above.

When comparing the total number of packets between QSP and ZMQ, we see that the ZMQ-based approach produces fewer packets in most scenarios. This is due to the straightforward pub/sub approach of ZMQ. While information about changed chunks is pushed to all subscribers with ZMQ, QSP requires additional communication overhead for traversing the quadtree, and thus more packets for synchronization are generated. However, network-level multicast partially amends for QSP's communication overhead. In the 16-server data center scenarios, the number of required data packets is consistently lower when comparing QSP to ZMQ. Besides, when focusing on security, QSP offers packet-level security that ensures authenticity and integrity of delivered data. The connection-oriented ZMQ implementation, however, does not provide such mechanisms. To ensure integrity and authenticity, connections need to be secured by e.g. adding SSL. While security mechanisms in NDN are added to every packet once, the ZMQ implementation requires to add security mechanisms on a connection-level, meaning that the effort increases with server cluster size.

When connecting clients to an online game, low latency for client–server communication is vital and may directly influence players' satisfaction. A player's client is served by the single server instance, which is responsible for the area in which a player's avatar is located. Therefore, the low latency requirement holds for the connection between client and server, whereas latency for the synchronization of the server cluster can be seen as less critical. While relaxed latency requirements hold for GSS, convergence time is an important aspect for sync protocols in general, which is why a convergence time study is conducted in the following. For measuring the convergence time, a server cluster consisting of four sync participants is emulated. In this server cluster, a single sync participant $P0$ changes map chunks in its region in constant intervals. The delay until all other sync participants ($P1$-$P3$) receive those changes is referred to as convergence time. We assume that the number of changed map chunks impacts the convergence time. This is why we vary the number of changed map chunks per update from 10 to 500. The results of this evaluation are presented in Fig. 12. The network links in this evaluation are configured with a one-way delay of 10 ms,

[11] https://github.com/PrismarineJS/mineflayer, last accessed: 2020-05-13.

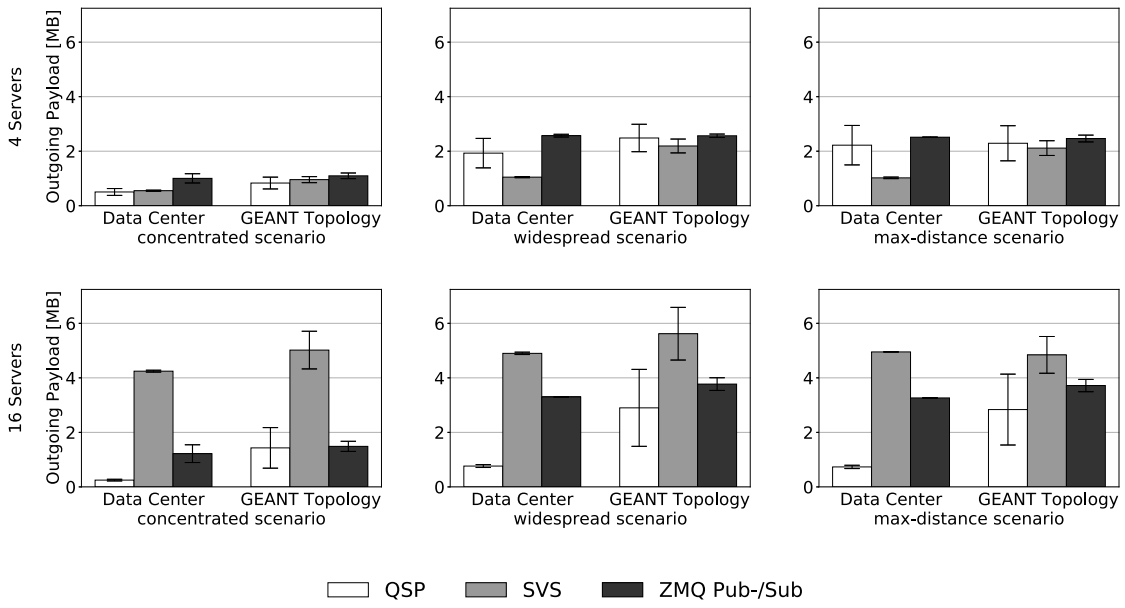


Fig. 10. Accumulated sync packet payload sent by the sync participants' nodes. Error bars depict 95% confidence intervals resulting from six emulation runs.

resulting in a lower bound for convergence time of 20 ms.[12] Every boxplot is based on the measurements of 60 updates of *P*0's region, where updates consists of 10 to 500 map chunk changes.

As expected, the push-based ZMQ implementation performs best with a convergence time close to the lower bound. ZMQ is a framework intended for low-latency communication and is built upon the well-proven TCP/IP stack. The used NDN stack,[13] however, is a reference implementation originating from research projects and not yet optimized for fast packet processing, leading to an added latency for the NDN approaches. In the NDN-based SVS protocol, *P*0 pushes notifications of dataset changes utilizing multicast sync requests to the other sync participants. After being notified, the other sync participants fetch the actual changes via Interest/Data exchanges. This process leads to a theoretical lower bound of 1.5 RTT (60 ms) for SVS. In the evaluation, the NDN stack and the SVS implementation result in a convergence time of approx. 150 ms. The NDN-based QSP is designed as a pull-based protocol and its convergence time depends on the number of map chunk changes. Due to the pull behavior, the sync participants are not notified about changes, which might increase the convergence time. Besides, when the number of changes is larger than the SHR threshold (used SHR threshold: 200, cf. Section 3.5), multiple request/response iterations are required to deliver map chunk changes. Focusing on the convergence times when less than 200 map chunks are changed, QSP's convergence time is as good or even lower as compared to SVS. Revisiting QSP's design, only one request and response is required to deliver those changes, where the request can be emitted prior to the data generation, leading to a lower bound for the convergence time of 0.5 RTT (20 ms). In practice, the convergence time depends on the

[12] Individual servers are connected via a central node resulting in a distance of two hops between two sync participants. This results in a one-way delay of 20 ms and a round-trip time (RTT) of 40 ms.

[13] Named Data Networking Forwarding Daemon (NFD v0.6.1), https://github.com/named-data/NFD.

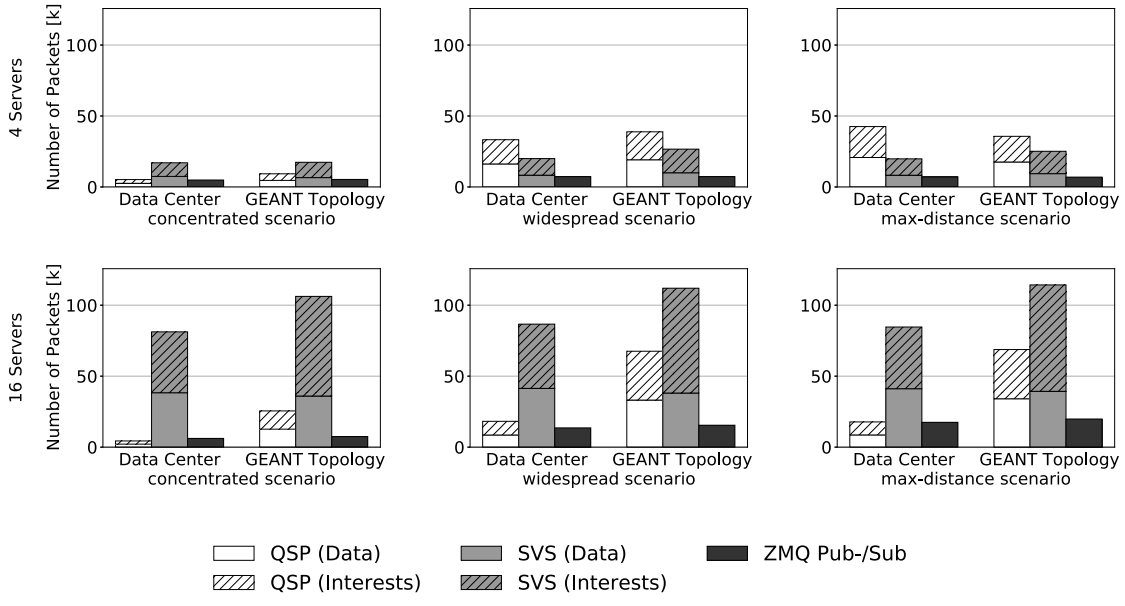


Fig. 11. Number of received interest and sent data packets by sync participants (accumulated) in different server cluster setups and client movement scenarios.

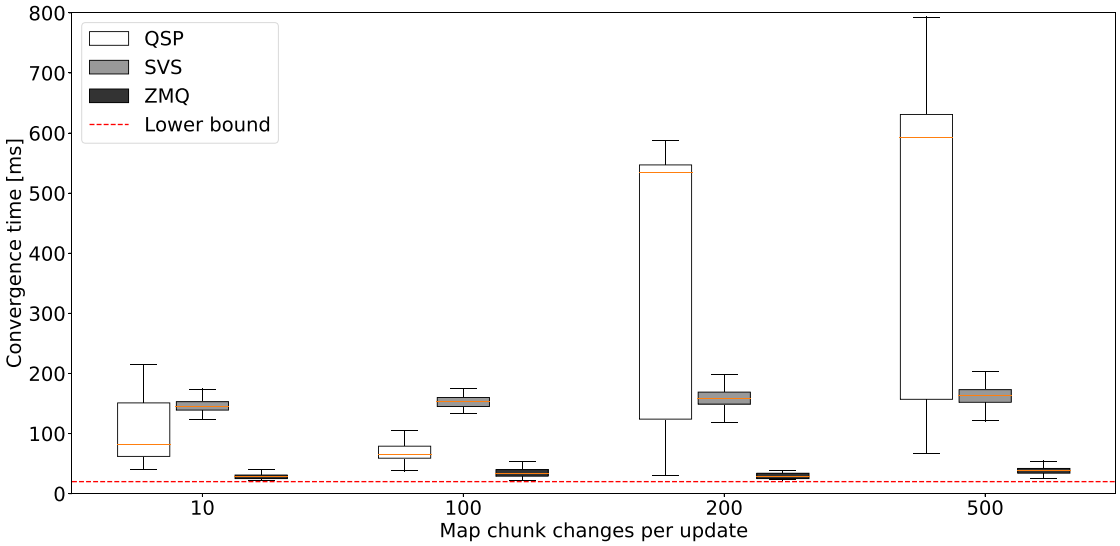


Fig. 12. Convergence time: time interval between one sync participant changing map chunks in its region and all subscribed participants having received the changes to the region.

time when sync requests are sent, which is not synchronized with the generation of the data. This leads to strong variations of the measured convergence time for QSP. With 200 or more changed map chunks, a single request/response cycle is not sufficient to retrieve changes, leading to a significantly higher convergence time. However, we want to highlight two simple engineering strategies that might reduce the convergence time of QSP: (1) instead of sending periodic sync requests to retrieve current changes, sync requests could be issued to retrieve future changes and kept pending at the producer. This strategy – also known as *Interest pipelining* [18] – introduces a push-like behavior for QSP. (2) To prevent exceeding the SHR threshold, the initial sync requests can already target lower quadtree levels for high activity regions and thereby eliminate the need of follow-up requests that delay convergence.

The presented figures in this section show that QSP is able to keep up with state-of-the-art protocols when used for GSS. However, the main appeal of QSP lies beyond performance gains: it simplifies architectural issues found in current systems. By decoupling the game world from sync participants, QSP does not require any information about other sync participants. SVS and ZMQ establish direct communication between all sync participants. This is implemented by addressing the other participants via IP addresses or by issuing interests including name prefixes that uniquely identify the other sync participants. In contrast, QSP issues sync requests for map regions and no matter which sync participant is hosting that region, the network forwards the requests correctly. QSP enables us to build systems lying beyond conventional communication patterns. In the next section, we demonstrate how QSP is used to synchronize a game server cluster, in which sync participants have no knowledge of others in the cluster.

5. Moving from server-based to region-based synchronization

In the previous section, a server cluster use case for GSS was discussed. In a server cluster, every sync participant is responsible for a certain region. During the sync process, participants issue their requests for exactly the regions that are managed by the other participants. In large server clusters, this results in a high number of remote regions that need to be synchronized. This high number of remote regions is partially caused by our current notion of game servers.

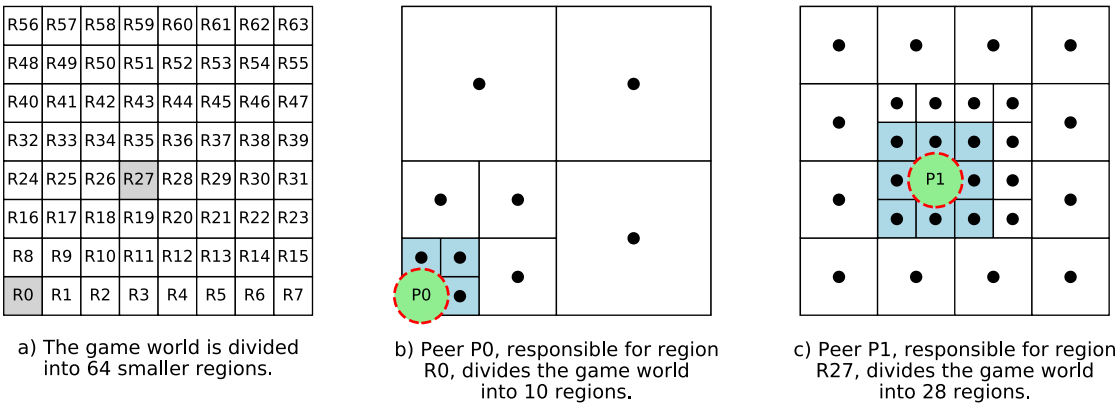
Focusing on the need of an individual server simulating a certain part of our Minecraft game world, we see that the server does not need knowledge of the other servers. For the simulation of a region, a server only needs to know what the game world outside of its region looks like. Also, not the whole game world is of equal importance to the server. A character moving on the other end of the world has lower priority than an arrow suddenly entering the server's region. This geographic context of data cannot be easily considered with current protocols but becomes possible by utilizing application-level information in the naming scheme, as done by QSP. Adding the quadtree structure as geographic context to NDN names allows region-based synchronization. Thereby, the synchronization of certain areas can be prioritized while removing the coupling between map regions and game servers.

5.1. Region-based game world structuring

Leaving the notion of the server cluster and of game servers aside allows to see the situation in a different light. When focusing on an individual peer who wants to simulate a specific region of the game world, instead of thinking of servers that want to jointly simulate the game world, the scenario as a whole stays more or less the same—with the exception that the restrictions given by the servers' regions disappear.

The role of a sync participant now is to simulate the region it is interested in. For simulating that region, the sync participant needs information about the remote regions. The regions geographically close to its own region are more important for the simulation, but still, a picture of the complete game world is needed.

The game world depicted in Fig. 13(a) can be divided into 64 smaller regions. In the server cluster use case presented in Section 4, the world would be jointly simulated by 64 individual sync participants, each responsible for a single region. Every sync participant would request changes from all remote regions, summing up to requests for 64×63 remote regions in the whole cluster. In the region-based use case presented in this section, a sync participant is mainly interested in simulating its own region and when requesting remote regions, the sync participant has a focus on parts of the world bordering its own region. Using QSP's quadtree, the game world can be divided in a way that prioritizes neighboring regions that are more important for the sync participant's game simulation. Bordering regions are requested as small individual regions, similar to how it would be done in the server cluster use case. Regions that are further away in the game world are



a) The game world is divided into 64 smaller regions.

b) Peer P0, responsible for region R0, divides the game world into 10 regions.

c) Peer P1, responsible for region R27, divides the game world into 28 regions.

Fig. 13. Reduction of the number of remote regions by quadtree-based structuring in the region-based QSP mode. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

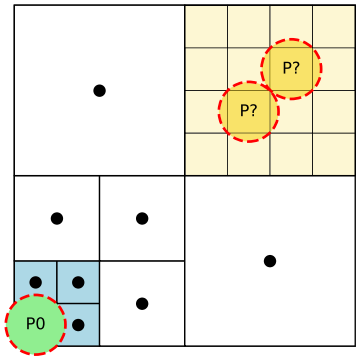


Fig. 14. Larger remote regions hide individual smaller regions that are managed by multiple individual sync participants.

summarized in larger regions by stepping up to a higher quadtree level. Thereby, the number of remote regions and the number of required sync requests is reduced. Two examples for structuring the world based on a sync participant's region are visualized in Fig. 13(b) and (c). Peers *P0* and *P1* are simulating regions *R0* and *R27*, respectively; neighboring regions are highlighted in blue. To synchronize the whole game world, individual small regions are summarized to larger regions by using the quadtree and only sync requests for regions marked with a circle are issued. In the case of *P0*, the game world is divided into only ten regions, meaning that only nine (instead of 63) remote regions need to be synchronized.

5.2. The relation between sync participants and remote regions

The reduction of remote regions, as discussed in the previous section, means that sync requests for regions that are simulated by multiple sync participants are sent. Focusing on Fig. 14, a single sync request for the whole region in the right upper corner is issued by *P0*. Realistically, there is a chance that the large requested region is jointly simulated by multiple individual sync participants, denoted as *P?*. Since those sync participants *P?* also synchronize their regions using QSP, each of those sync participants has information about the whole larger region and can thereby respond to the sync requests by *P0*.

Looking at NDN's forwarding plane, a sync request for a region that is larger than a participant's actual region needs to be forwarded to any other participant that can handle the request. A sync participant *P?* responsible for the region `/prefix/1/1/2` is able to answer requests for its own region `/prefix/1/1/2` as well as for the larger regions `/prefix/1/1` and `/prefix/1`. These region names are used by NDN to forward sync requests to appropriate participants and are therefore populated to the forwarding information bases (FIB) of all nodes in the network. For the population of FIB entries either NDN routing protocols or static FIB entries can be used.[14] When an interest for one of those names is processed, NDN's forwarding plane decides to which sync participant the interest is forwarded to.

Focusing on the protocol details of QSP presented in Section 3.4, we know that the amount of SHRs should be kept low, the amount of CDRs high. A CDR contains versions of map chunks changed since the last sync request. For deciding which map chunks changed, the region owner uses the requester's hash value of the requested region. Considering that larger regions are collaboratively simulated by multiple sync participants and that the hash value of the larger region results from hashing all child regions that are simulated by different sync participants, an issue may be arising here. With perfect GSS, every sync participant would know the latest version of every map chunk immediately. Hence, the larger region's hash would be the same for all sync participants. In reality, GSS causes a synchronization delay, leading to sync participants having different versions for some map chunks resulting in different hash values for the larger area. This possible hash mismatch increases the demand for SHR responses and increases the total number of required sync requests. While the efficiency of the QSP might be decreased, the protocols' functionality, however, sustains. A thorough evaluation of the region-based QSP operation is found in the next section.

5.3. Demonstration of region-based QSP

Similar to the server-mode evaluation, we construct a prototype for GSS in a region-based setting. The simulation of the game world in this evaluation is distributed to 16 sync participants, each of which simulates equally sized regions of the game world. Instead of requesting changes from regions that are simulated by individual other sync participants, remote regions are structured using QSP's quadtree as depicted in Fig. 13.

Comparing the results of the region-based prototype to server-based synchronization in Fig. 15, no major differences concerning the

[14] For more details, we want to refer to our discussion dealing with forwarding scalability in Section 3.6. However, routing and forwarding in NDN are different research fields not in the focus of this paper. A thorough discussion of routing and forwarding in NDN is provided in [19].

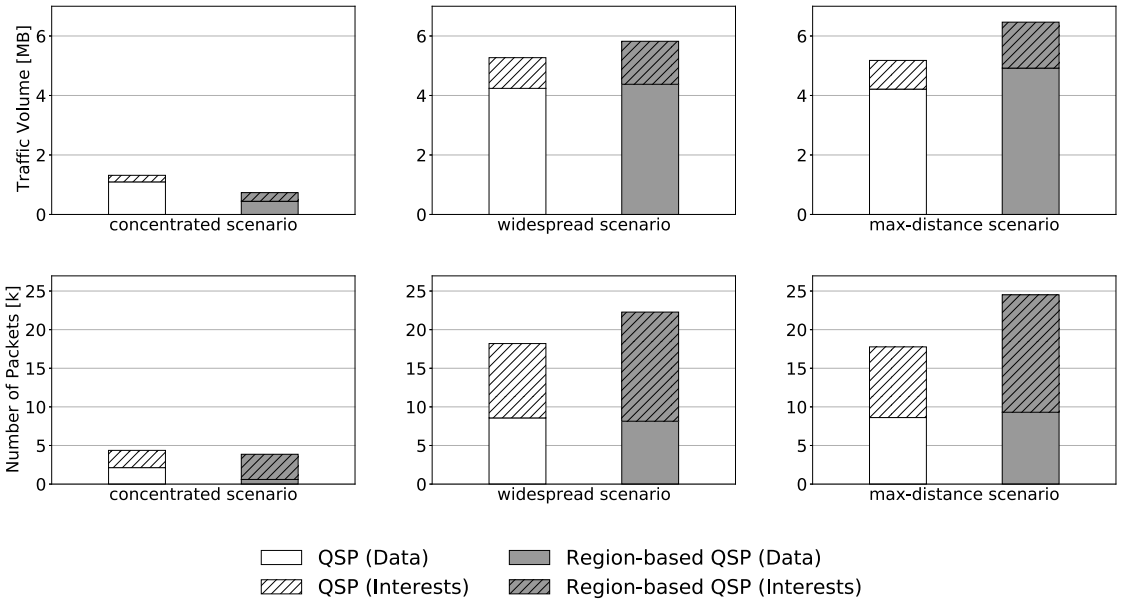


Fig. 15. Comparison of QSP in server-based and region-based mode. The figure includes traffic volume and packet counts for incoming interests and outgoing data packets on the sync participants' nodes.

produced traffic volume are observable. Focusing on the concentrated client movement scenarios (leftmost charts), we see that the region-based approach reduces both the amount of sent bytes and the number of required packets. This is the result of the map chunk changes being concentrated to a small part of the quadtree and a large number of sync participants simulate regions with only low, or without client activity. When sending sync requests for larger regions, it might happen that only a few or no changes happened in the region and hence no hash-mismatch occurs. Thus, by reducing the number of sync regions, the total number of required sync requests also is successfully reduced. In the other scenarios, client activity is distributed to larger parts of the quadtree leading to fewer regions without change and more frequent hash mismatches. The advantage given by the lower number of remote regions is mitigated by the higher number of SHR responses caused by these hash mismatches. This leads to about the same amount of required sync requests for region-based and server-based QSP in the widespread and max-distance scenarios.

This evaluation demonstrates QSP's ability to synchronize game state information of a game world based on the needs of a sync participant. Close regions are more important for the game's simulation and are requested with fine granularity. More distant areas are summarized in a single request, no matter how many other sync participants are responsible for the requested region. The responsibility for finding the requested data is shifted from the application to the network layer and decouples map regions from game servers. Thereby, it becomes possible for game servers to work independently without knowing other region assignments. This means that the need for exchanging that information is eliminated and the management of the server cluster becomes easier. Scaling the simulation of a region from one to multiple servers during operation does, for instance, not influence the sync process and GSS continues to work without further actions required.

Thinking beyond conventional game architectures, the presented region-based synchronization mode of QSP could allow for building P2P gaming architectures. A player who wants to play Minecraft could self-host the region it is playing in and synchronize its region with peers playing in surrounding regions without knowing addresses or other identifiers of those peers. The only binding restriction is that no two players can host the same part of the world. When this is ensured in a bootstrapping phase, the NDN network layer would forward sync requests to the corresponding peers, and QSP could allow thousands of players to play in the same distributed world. Besides, NDN's information-centric nature and packet-level security could support a higher availability of the distributed world. Even if a peer goes offline, data produced by the peer is accessible via in-network caches; the signature added when publishing data ensures integrity. However, players regularly leaving and joining the game – referred to as churn in the P2P context – might require additional services to allow for faster reorganizing of map regions. Even though players leaving the game unexpectedly do not influence the synchronization process negatively, other players' satisfaction could suffer in regions where the game simulation suddenly stopped. As discussed in the next section, a thorough evaluation of region-based QSP for P2P systems is planned as part of our future work.

6. Conclusion

In this paper, we focused on inter-server game state synchronization (GSS) required for online games that are distributed to large server clusters. While conventional systems for GSS require knowledge of the server cluster topology, i.e. information about which servers are part of the server cluster and which servers simulate which regions, our proposed solution works without this knowledge. Geographic relations between objects in the game world are exploited to build a system that decouples the game world from the individual game server cluster instances. In the proposed Quadtree Synchronization Protocol (QSP), a quadtree is used to structure the game world hierarchically in regions of varying sizes and thereby allows achieving high flexibility. A server responsible for a certain region can prioritize the synchronization of close-by regions by requesting changes of those regions in fine granularity. A region further away and thereby having a lower influence on the server's region does not need to be requested with fine granularity. Hence multiple remote regions can be summarized in a single sync request. No matter if those regions are hosted by a single server or multiple servers, the NDN network layer, which is the basis for our protocol, forwards the request to a suitable node so that the synchronization succeeds without relying on a specific server instance.

Besides providing server independence and thereby increasing fault tolerance as well as easier recovery after a server failure, QSP provides security on a packet level. When distributing updates in connection-based systems, such as in TCP-based protocols, connections are secured, not the sent content. For instance, in a server cluster with 16 servers, the same update is sent over 15 unique connections—one connection to

every other server. Measures for providing confidentiality and integrity need to be in place for every connection, meaning that calculation of checksums, creation of signatures, and encryption is done multiple times. In NDN, however, the update message itself is secured once and no matter how many participants receive the update, the workload stays the same. Besides, the possibility to verify the integrity of every single packet allows using features such as network-level multicast or in-network caching, without the need to worry about security.

The main advantage of QSP is architectural. Besides discussing these advantages, we demonstrate the performance of QSP by network emulations. Our first evaluation shows that QSP can hold up against an IP-based implementation and another NDN-based sync protocol. Our second evaluation demonstrates the region-based sync mode, where sync participants do not need to know which other sync participants exist and which regions are hosted by which other participants. This region-oriented sync feature is novel in QSP and such behavior cannot be provided by other current sync protocols.

While QSP was demonstrated for GSS in this paper, we want to highlight possible other use cases. We argue that many networked applications dealing with geographically organized data can make use of QSP, among them GIS applications and AR/VR mirror worlds. Current implementations of such services rely on servers which collect and maintain data of every entity contributing data to the application. Besides the high computational requirements and bandwidth demands of such servers, they represent a single point of failure and a possible performance bottleneck. Using QSP for such applications shifts the storage and management of data to a distributed P2P-like cluster and hence removes the aforementioned drawbacks. Every node contributing data keeps its local copy of the quadtree, whereby the state of the application gets distributed. Also, since every piece of information is signed by the contributing peer, integrity is ensured by every single peer instead of controlled by a server instance, increasing trust in the overall system.

In future work, we plan to improve QSP's synchronization delay by integrating concepts of the real-time data retrieval protocol presented in [18]. QSP currently issues sync requests in periodic intervals that cause an inherent synchronization delay. Switching from periodic sync requests to a real-time data retrieval approach removes the inherent synchronization delay and reduces the synchronization latency to the one-way link delay, as found in push-based solutions.

Moreover, we identified the potential of using QSP in P2P online gaming architectures, as discussed in Section 5.3. Integrating QSP in a real-world game and converting it to a P2P architecture could give valuable insights concerning the practicability of NDN-based P2P gaming architectures. However, the development of additional services, like a bootstrapping service that ensures that peers do not simulate overlapping regions, is required. Such a prototype gaming application working on a global scale, e.g. via the global NDN testbed,[15] could provide insights for other research areas as well. This means, for instance, different approaches for providing better forwarding scalability, as discussed in Section 3.6, could be evaluated in a real-world application.

Lastly, we want to highlight the applicability of the quadtree naming scheme for other use cases. Converting the quadtree to an octree allows inferring names for three-dimensional spaces. Focusing on recent developments in multimedia streaming, a high interest in volumetric video streaming (a.k.a. point cloud streaming [20,21]) is seen. In volumetric videos, additional depth information allows representing 3D objects. For streaming volumetric videos, immense bitrates are required. We argue that NDN's network-level multicast functionality could reduce inefficiencies when streaming such content to multiple consumers. Besides, applying the octree naming scheme could allow consumers selecting different quality representations for the streamed objects based on their location.

Source code resulting from this work is published as open-source software and available in an online repository.[16]

[15] https://named-data.net/ndn-testbed/, last accessed: 2020-11-03.

[16] https://github.com/phylib/QSPArtifacts.

CRediT authorship contribution statement

Philipp Moll: Conceptualization, Methodology, Software, Investigation, Visualization, Writing - original draft. **Selina Isak:** Software, Visualization. **Hermann Hellwagner:** Conceptualization, Methodology, Writing - review & editing. **Jeff Burke:** Conceptualization, Methodology, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Entertainment Software Association, 2019 Essential facts about the computer and video game industry, 2020, URL: https://www.theesa.com/esa-research/2019-essential-facts-about-the-computer-and-video-game-industry/, (accessed 2020-03-31).

[2] P. Moll, M. Lux, S. Theuermann, H. Hellwagner, A network traffic and player movement model to improve networking for competitive online games, in: Proceedings of the 16th Annual Workshop on Network and Systems Support for Games (NetGames 2018), 2018, pp. 1–6, URL: https://dl.acm.org/citation.cfm?id=3307315.

[3] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, kc. Claffy, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang, Named data networking, ACM SIGCOMM Comput. Commun. Rev. (ISSN: 0146-4833) 44 (2014) 66–73.

[4] P. Moll, S. Theuermann, N. Rauscher, H. Hellwagner, J. Burke, Inter-Server game state synchronization using named data networking, in: Proceedings of the 6th ACM Conference on Information-Centric Networking (ICN' 19), ACM, New York, NY, 2019, pp. 12–18, http://dx.doi.org/10.1145/3357150.3357399.

[5] E. Cronin, A.R. Kurc, B. Filstrup, S. Jamin, An efficient synchronization mechanism for mirrored game architectures, Multimedia Tools Appl. (ISSN: 13807501) 23 (1) (2004) 7–30, http://dx.doi.org/10.1023/B:MTAP.0000026839.31028.9f.

[6] S.-Y. Hu, C. Wu, E. Buyukkaya, C.-H. Chien, T.-H. Lin, M. Abdallah, J.-R. Jiang, K.-T. Chen, A spatial publish subscribe overlay for massively multiuser virtual environments, in: 2010 International Conference on Electronics and Information Engineering, Vol. 2, IEEE, 2010, pp. V2–314.

[7] P. Moll, S. Theuermann, H. Hellwagner, J. Burke, Distributing the game state of online games: Towards an NDN version of minecraft, in: 2019 IEEE International Conference on Communications Workshops (ICC Workshops), 2019, pp. 1–6.

[8] Z. Zhu, A. Afanasyev, Let's chronosync: Decentralized dataset state synchronization in named data networking, in: 2013 21st IEEE International Conference on Network Protocols (ICNP), 2013, pp. 1–10.

[9] W. Shang, A. Afanasyev, L. Zhang, NDN-0056: VectorSync: Distributed Dataset Synchronization over Named Data Networking, Tech. Rep., 2018, URL: https://named-data.net/publications/techreports/ndn-0056-1-vectorsync/.

[10] T. Li, Z. Kong, S. Mastorakis, L. Zhang, Distributed dataset synchronization in disruptive networks, in: 16th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (IEEE MASS), 2019, p. 10, http://dx.doi.org/10.1109/MASS.2019.00057.

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

[12] A.K.M.M. Hoque, S.O. Amin, A. Alyyan, B. Zhang, L. Zhang, L. Wang, NLSR: named-data link state routing protocol, in: Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking - ICN '13, ACM, New York, New York, USA, 2013, p. 15, http://dx.doi.org/10.1145/2491224.2491231.

[13] A. Guttman, R-Trees: A dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 1984, pp. 47–57, http://dx.doi.org/10.1145/602259.602266.

[14] R.C. Merkle, A digital signature based on a conventional encryption function, in: Advances in Cryptology — CRYPTO '87, Springer Berlin Heidelberg, Berlin, Heidelberg, 1988, pp. 369–378, http://dx.doi.org/10.1007/3-540-48184-2_32.

[15] Y. Zhang, Z. Xia, A. Afanasyev, L. Zhang, A note on routing scalability in named data networking, in: 2019 IEEE International Conference on Communications Workshops (ICC Workshops), 2019, pp. 1–6, http://dx.doi.org/10.1109/iccw.2019.8756677.

[16] A. Afanasyev, C. Yi, L. Wang, B. Zhang, L. Zhang, SNAMP: Secure namespace mapping to scale NDN forwarding, in: 18th IEEE Global Internet Symposium, 2015, http://dx.doi.org/10.1109/INFCOMW.2015.7179398.

[17] A. Afanasyev, X. Jiang, Y. Yu, J. Tan, Y. Xia, A. Mankin, L. Zhang, NDNS: A DNS-like name service for NDN, in: 26th International Conference on Computer Communications and Networks, ICCCN 2017, 2017, http://dx.doi.org/10.1109/ICCCN.2017.8038461.

[18] S. Mastorakis, P. Gusev, A. Afanasyev, Real-time data retrieval in named data networking, in: Proceedings of 2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN 2018), IEEE, 2018, pp. 61–66, http://dx.doi.org/10.1109/HOTICN.2018.8605992.

[19] C. Yi, J. Abraham, A. Afanasyev, L. Wang, B. Zhang, L. Zhang, On the role of routing in named data networking, in: ICN 2014 - Proceedings of the 1st International Conference on Information-Centric Networking, 2014, pp. 27–36, http://dx.doi.org/10.1145/2660129.2660140.

[20] M. Hosseini, C. Timmerer, Dynamic adaptive point cloud streaming, in: Proceedings of the 23rd Packet Video Workshop, 2018, pp. 25–30, http://dx.doi.org/10.1145/3210424.3210429.

[21] J. van der Hooft, T. Wauters, F. De Turck, C. Timmerer, H. Hellwagner, Towards 6DoF HTTP adaptive streaming through point cloud compression, in: Proceedings of the 27th ACM International Conference on Multimedia, ACM, 2019, pp. 2405–2413, http://dx.doi.org/10.1145/3343031.3350917.

Philipp Moll is a PreDoc Scientist at the Institute of Information Technology (ITEC) at Klagenfurt University. He is currently pursuing a Ph.D. in the field of computer networking. His main research interest is bringing information-centric networking architectures to online gaming, but he is also involved in work about reproducibility in research, multimedia communications, and computer games. Philipp Moll received his M.Sc. in computer science from Klagenfurt University in 2016.

Selina Isak is a Student Research Assistant at the Institute of Information Technology (ITEC) at Klagenfurt University. She is currently pursuing a B.Sc. in Applied Computer Science at Klagenfurt University. In the course of her work, she is involved in information-centric networking research and multimedia communication.

Dr. Hermann Hellwagner is a full professor of Informatics in the Institute of Information Technology (ITEC), Klagenfurt University, Austria, leading the Multimedia Communications group. His current research areas are distributed multimedia systems, multimedia communications, information-centric networking, and communication in multi-drone systems. He has received many research grants from national (Austria, Germany) and European funding agencies as well as from industry, is the editor of several books, and has published more than 250 scientific papers on parallel computer architecture, parallel programming, and multimedia communications and adaptation. He is a senior member of the IEEE, member of the ACM and OCG (Austrian Computer Society); he was Vice President of the Austrian Science Fund (FWF).

Jeff Burke is Professor In-Residence of performance and technology and Associate Dean, Technology and Innovation at the UCLA School of Theater, Film and Television. His research explores the intersections of the built environment, computer networks, and storytelling. He cofounded REMAP, a joint center of TFT and the Henry Samueli School of Engineering and Applied Science, which uses a mixture of research, artistic production, and community engagement to investigate the interrelationships among culture, community, and technology. From 2006–2012, Burke was the area lead for participatory sensing at the NSF Center for Embedded Networked Sensing (CENS). Burke was co-PI and application lead for the Named Data Networking research project, a 12-campus effort supported by the NSF Future Internet Architecture program, and is part of the management team for the NIST Named Data Networking Consortium started in 2020.