Multiplayer Online Game Communication Using Named Data Networking

Stefano Lupo, lupos@tcd.ie, *MAI, Trinity College Dublin*

***Abstract*—NDN is a new Internet architecture in which the core abstraction changes from sending data between two specific hosts in a network, to naming self-contained pieces of data, and allowing that data to be fetched by the name alone. NDN aims to replace IP as the *universal network layer* of tomorrow's Internet, and supports running over a variety of lower level protocols such as Ethernet, Bluetooth and even IP itself. With the ever-increasing popularity of MOGs, this research aimed to investigate the possibility of using NDN as the core communication protocol for a multiplayer game.**

As the data generated in MOGs is so varied, one of the main goals of this research was to determine how well NDN can support all of these different data types, some of which fit nicely into a content-oriented abstraction, and some of which are entirely host oriented. The other major challenge associated with MOGs is that they require extremely high-performance networking solutions that provide high bandwidth and low latency.

The research led to the development of a novel protocol for synchronizing remote game objects across all players in the game using NDN. This protocol was designed to exploit the three main benefits of NDN - *interest aggregation*, *native multicast* and *in-network caching*. A peer-to-peer 2D game was implemented and used to test the performance of the synchronization protocol. The results show that NDN is an excellent choice for MOG communications and that the developed synchronization protocol could easily support 16 concurrent players, while providing the performance required for a fast paced MOG.

***Index Terms*—Multiplayer games, Named Data Networking, Peer-to-peer, Dataset Synchronization**

I. Introduction

Today, most devices make use of the so called Internet Protocol (IP) as the primary mechanism for global communication. The design of IP was heavily influenced by the success of the 20th century telephone networks, resulting in a protocol tailored towards point-to-point communication between two hosts. IP is the *universal network layer* of today's Internet, which implements the minimum functionality required for global interconnectivity. This represents the so called *thin waist* of the Internet, upon which many of the vital systems in use today are built [1]. The design of IP was paramount in the success of the modern day Internet. However, in recent years, the Internet has become used in a variety of new non point-to-point contexts, rendering the inherent host based abstraction of IP less than ideal.

The Named Data Networking (NDN) project is a continuation of an earlier project known as Content-Centric Networking (CCN) [2], both of which are instances of a broader networking architecture known as Information Centric Networking (ICN). The CCN and NDN projects represent a shift in how networks are designed, from the host-centric approach of IP to a data centric approach. NDN provides a new global communication mechanism, maintaining many of the key features which made IP so successful, while improving on the shortcomings uncovered after three decades of use. The design of NDN aligns with the *thin waist* ideology of today's Internet and NDN strives to be the universal network layer of tomorrow's Internet.

Multiplayer online games (MOGs) have become more and more popular over the past several decades and are now a widely enjoyed pastime amongst people of all ages. The complexity of MOGs has also risen dramatically in that time, with modern MOGs providing huge immersive game worlds which can support thousands of concurrent players. As games are realtime in nature, modern MOGs require extremely high performance networking solutions to support large numbers of players.

This research aims to bring these two fields together by examining the use of NDN as the primary communication mechanism for MOGs. Although substantial research was carried out in order to gain a deep understanding of each of the fields, the main focus of the project was to design and implement a real NDN based MOG and to build a comprehensive framework for testing the game in a variety of scenarios.

Finally, all of the source code developed for this project is available on GitHub at the following URL: *github.com/stefano-lupo/ndn-thesis*.

II. State of the Art

A. Named Data Networking (NDN)

NDN uses two core primitives - *Interest* packets and *Data* packets. In order to request a piece of data from the network, an Interest packet is sent out with the name field set to the name of the required piece of data.

NDN requires three key data structures to operate - a *Forwarding Information Base (FIB)*, a *Pending Interest Table (PIT)* and a *Content Store (CS)*.

The FIB is used to determine which interface(s) an incoming Interest should be forwarded upstream through. This is similar to FIB used on IP routers. The PIT stores the names of Interests and the interface on which the Interest was received, for Interests which have been forwarded upstream, but not yet had any Data returned. The CS is used to cache Data packets received in response to Interests expressed. The CS allows any NDN node to satisfy an interest if it has the corresponding Data packet, even if it is not the producer itself.

On receipt of an Interest, the CS is checked to see if there is a cached copy of the Data corresponding to the name in the

Interest. If a copy exists with the appropriate freshness, the Data packet can be sent back over the requesting Face and the Interest packet is satisfied. Thus, NDN provides **in-network caching**

If there is no cached copy of the Data in the CS, the PIT is then checked. If a PIT entry containing the Interest name exists, this indicates that an equivalent Interest packet has already been seen and forwarded upstream. In this case, the Interest packet is **not forwarded upstream** a second time. Instead, the requesting face is added to the list of downstream faces in the PIT entry. This list of faces represents the downstream links which are interested in a copy of the Data. This is known as **Interest aggregation**.

If there is no PIT entry, the FIB is then queried to extract the next hop information for the given Interest. If there is no next hop information, a NACK is typically returned. If an FIB entry is present, a PIT entry for the given Interest is created and the packet is forwarded upstream, using the next hop contained in the FIB entry.

On receiving a Data packet, the PIT is checked to ensure that the Data packet had actually been requested. If there is no PIT entry, the node never expressed or forwarded an Interest for this piece of data. This means the Data packet is unsolicited and is typically dropped.

Otherwise, the Data packet is sent over all of the requesting faces contained in the PIT entry, the PIT entry is removed and the Data is added to the CS. Thus, a single Data packet published by a producer can be sent to multiple consumers, meaning NDN supports **native multicast**.

B. Dataset Synchronization (DS) in NDN

A common requirement in distributed, P2P environments is for nodes to read and write to a shared dataset. In order to provide all participants with a common view of the dataset, a *dataset synchronization protocol (DSP)* is required. The importance of DSPs is amplified in an NDN context as most applications are developed with a distributed P2P architecture in mind. This is done to enable high scalability through the exploitation of the features offered by NDN such as in-network caching and native multicast. As such, a lot of research into the area of DS in NDN has been conducted and one of the goals of the research is to abstract away the need for NDN application developers to consider DS.

Traditionally, IP based solutions for DS take one of two approaches - centralized or decentralized.

Alternatively, a decentralized approach can be taken in which all nodes communicate with one and other. In an IP based solution, this requires each node to maintain $n - 1$ connections to every other node, for example using a TCP socket. This approach mitigates the problem of having a bottleneck in the system, resulting in a more scalable solution, at the cost of requiring a considerably more complex protocol in order to maintain a consistent view of the dataset amongst all nodes.

Decentralized approaches are typically more scalable than centralized approaches, though the scalability of the decentralized approach is limited by the connection oriented abstraction of IP, as the number of connections required scales quadratically with the number of nodes. The data oriented abstraction of NDN overcomes this issue as nodes are no longer concerned with *who* they communicate with and are instead concerned with producing and consuming named pieces of data, which can be fetched from and published to the network. The decentralized approach also benefits from the features offered by NDN such as interest aggregation, native multicast and in-network caching.

NDN can achieve distributed DS by synchronizing the namespace of the shared dataset among a group of distributed nodes [3]. Several protocols have been developed to achieve this including CCNx Sync 1.0 [4], iSync [5], ChronoSync [6], RoundSync [7] and PSync [8]. Although many of these existing DSPs are well developed, they are not suitable for use in a MOG environment as they only inform nodes of updates to the dataset, and require a second Interest / Data exchange to fetch the updates. Due to the low latency required for MOGs, a protocol which retrieves updates in a single Interest / Data exchange is required.

C. Multiplayer Online Games (MOGs)

On a fundamental level there are only two architectures to choose from, *Client/Server (C/S)* or *Peer-to-Peer (P2P)*. However, there is a huge amount of variation within each of those architectures and even combinations of the two architectures such as *MultiServer (MS)*, which uses a small number of centralized servers to somewhat distribute the load, and *Hybrid* which uses both C/S and P2P elements. The trade-offs between C/S and P2P architectures in a MOG context are very similar to those found elsewhere in computer science. C/S architectures are simpler and more robust to cheating, but suffer from scalability issues and the opposite is true for P2P architectures.

MOGs typically follow a *primary copy* replication approach. For each game object (e.g. players and NPCs), there exists an authoritative *primary* copy, and this exists on one node only. All other copies are *secondary copies*, and are merely replicas of the primary copy. All updates to game objects are performed on **primary copies only**. The results of these update operations are then sent to all other players, who update their secondary copies accordingly [9].

In video games, the rate at which the local game world is updated and redrawn at is known as the *frame rate* and anything below 30 frames-per-second results in a poor user experience. Using the Internet as it stands today, consistently receiving remote updates at a rate of even 30Hz would be extremely challenging and unreliable due to packet loss, limited bandwidth, congestion and propagation times alone. The requirement for extremely frequent updates, coupled with the amount of remote game objects that need updating, means that moving remote game objects based on received updates alone is simply not feasible, and attempting to do so leads to very jittery movement. Dead reckoning (DR) is a commonly employed solution to this problem in which remote game objects are locally updated at a frequency higher than the rate of updates received for those game objects. As described by

Walsh, Ward and McLoone [10], *"DR is a short term linear extrapolation algorithm which utilises information relating to the dynamics of an entity's state and motion, such as position and velocity, to model and predict future behaviour"*. By including an entity's velocity as well as their new position in remote update packets, an extrapolated trajectory can be built for the game object, which defines their future position as a function of time. The local client can then move the remote game object along this trajectory in between actual remote updates, providing the appearance of smooth motion. Another interesting component of DR is that it can be used to dynamically control the rate at which updates are published. As all parties use the same extrapolation and convergence algorithms, the holder of the primary copy can also maintain a replica copy, which represents the extrapolated position as viewed by remote players. This mechanism can be used to dynamically control the rate at which updates are published depending on the motion characteristics of the game object. For example, if the game object is stationary, the extrapolated position over time will remain constant as the velocity vector is also zero. Thus, until the game object begins to move, there is no need to publish further updates. This can also apply to game objects moving on a constant trajectory, for example, a player moving due east in the game world.

Depending on the design of the MOG, players may only see a subsection of the game world at a given time. For example, in a top-down game or side scroller, the camera remains centred on the player's avatar and the player's viewport is a subregion of the entire game world. Similarly, in more complex 3D games, the player's view of the game world may be obstructed by objects such as rocks or trees, or the player may be inside of a building. The game world may also be divided into geographical regions such that a player's view of the game world is strictly limited to the geographical region they are currently in. In all cases, there is an opportunity to drastically reduce the amount of game objects that must be synchronized to present a consistent view of the game world to the player. This concept is defined in the literature as *interest management (IM)*. The most prevalent form of IM is *spatial IM*, in which only game objects that can be seen by the player are synchronized.

D. Closely Related Projects

Projects which focus on researching, designing and building MOGs using NDN as the communication mechanism are very relevant to the project. However, as NDN is still a relatively new technology in an early prototyping stage, only three projects were found in this area.

Egal Car [11] was the first investigation into building a MOG using NDN. Egal Car used an existing single player, Unity based, car racing game and focused on writing a P2P networking module for the game, allowing it to be played as a multiplayer game. The key limitation of Egal Car is that assets were not allowed to interact with one and other. Thus, the problem was simplified to one of DS, in which there is only one producer of content. Egal Car was also created in 2012 and used a framework which is no longer a part of the NDN platform. Finally, Egal Car was a proof of concept prototype, and there was no testing performed and there is no source code publicly available.

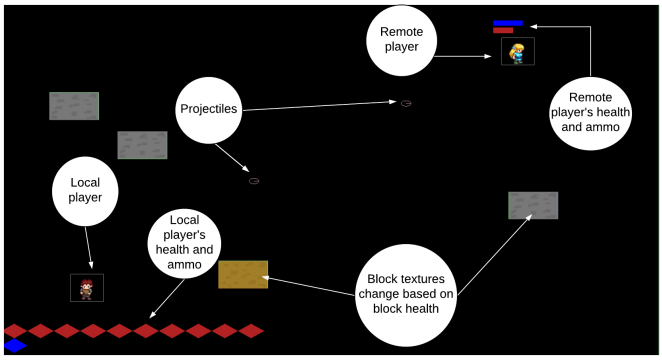

Figure 1. NDNShooter - a 2D, top down game developed to facilitate research into MOGs using NDN

Matryoshka [12] is another P2P MOG which runs over NDN. The core focus of Matryoshka was to come up with a way to partition the game world such that players would only be interested in other game objects in their partition. This was done by recursively partitioning the game world into 8 octants. In the implementation outlined, the partitioning was two layers deep, although this could be deeper for larger game worlds. The partition to which a game object belongs to is thus defined by two indices, representing the octant they are in at each of the two layers. Although an implementation is discussed, there is no source code available. There paper also lacks any results or evaluation section, indicating the architecture has not been tested.

NDNGame [13] describes the use of a hybrid architecture in which a conventional C/S approach using UDP over IP is used for the actual gameplay related networking, and NDN is used for the dissemination of the **game files**. The logic behind this approach is that the size of the initial files required to play the game are far larger than the packets which are sent when playing the game. The use of the conventional C/S architecture using UDP/IP is chosen due to the importance of network latency when playing the game. The paper's suggestion that using NDN in a MOG scenario is not feasible does not appear to be rooted in any actual testing or empirical evidence and thus, the main finding of the paper is only that NDN would be an ideal candidate for static game file dissemination.

III. NDNShooter - a 2D, Top Down, Shooting Game

NDNShooter was designed to facilitate research into how NDN performs in a MOG context and a screenshot from the game is shown in figure 1. NDNShooter contains both local and remote players, both of which can freely move around the game world and have their positions synchronized across all players. Players can place blocks in the game world, which are seen as yellow and grey rectangles in figure 1. Players can also interact with local and remote game objects by shooting projectiles at players and blocks and these interactions will be visible to all players in real time.

A. NDNShooter Data Taxonomy and Sync Protocols

One of the primary goals of the research was to characterize the different types of data found in modern multiplayer games. The first step to building a high-performance networking solution is to understand the different types of data required by the application and to characterize that data accordingly. The categories of data found in MOGs is highly influenced by the genre of the game. This research focuses on fast paced, real time games such as *first-person-shooters (FPS)* and *role-playing-games (RPGs)*, as opposed to *turn based* games, as these are substantially more challenging and interesting from a networking perspective. The design of NDNShooter was influenced by the MOG taxonomy to ensure all categories of data were represented.

The overall taxonomy of MOG data is shown in orange in figure 2, with the corresponding data in NDNShooter shown in blue, and the protocols used to synchronize the data in NDNShooter in white.

Static Content: MOGs make heavy use of data which is static and does not change over time. An example of this data would be textures for game world assets. Static content can also be configurable by players in the game world, for example, if players can design their own base or can use custom player sprite sheets. From a networking perspective, static content is an ideal candidate for caching. For example, if a player moves from one room to another, or requires the sprite sheet of a new player coming into view, the textures are likely to be cached by routers in the network, as other players would have previously required them. However, in comparison to the other categories of data, the frequency at which static content is requested from the network is so low that the ability to cache this data would likely have a negligible impact on the overall network performance. Due to the simplicity of game, there is not a lot of static content which needs to be sent over the network. Game world assets are packaged and shipped with the game. However, custom player spritesheets represent an ideal candidate for dissemination using NDN.

Realtime Streams: The second category of data found in MOGs is realtime data which is sent continuously, at a somewhat consistent interval. Due to the temporal consistency of this data, it is best considered as a stream. This is the data type which accounts for the majority of the network traffic and is usually the most critical in terms of game fluidity. The most common form of this data is due to the game reacting to player commands. However, this category can be further subdivided by the frequency at which this data is published. As players are free to roam around the world in NDNShooter, player position updates are required extremely frequently in order to provide the appearance of smooth motion of remote players. Players can also place blocks, though this ability is limited to once every two seconds. Thus, even if a player chooses to continuously places blocks at the maximum rate, the updates associated with block creation are still relatively low frequency in comparison to player position updates.

Non Synced: The third category of data found in MOGs is data which must be sent to remote players, but that does not change or need to be synchronized over time. Another key aspect of this data is that it is typically short lived. This data type can be thought of as events that occur in the game world as a result of player actions. Projectiles are extremely short lived in NDNShooter as they travel at a high speed. Once a projectile is produced, there are no further updates required for that projectile, aside from it being destroyed when it hits a player, block or the game world boundary. This is analogous to the event being consumed. Projectiles are created with an initial position and velocity and are then published to the network. On collision with the game world boundary, they are automatically destroyed locally by all players. However, on collision with a remote player, or a block created by a remote player, the projectile is destroyed and subsequent action is taken to inform other players. Thus, there is no requirement to synchronize projectiles over time, meaning they are essentially events published by a player and are either consumed by the player who created the projectile, or the player who interacts with the projectile.

Synchronized Datasets: The final category of data found in MOGs are distributed datasets which must be *strongly synchronized*. These are elements of the game which all players must agree on. An example of this data type is the state of the game world on a macro scale. This could range from which *non-playable characters (NPCs)* are alive and what path they are currently moving on, to what health kits are currently present in the game world. This data type is updated at a very low frequency, but requires strict consistency amongst game players and can therefore use more expensive protocols which would not be suitable for other data types. Player discovery is a good example of a dataset which needs to be synchronized across all game players. The rate at which updates are performed on this dataset is approximately equal to the rate at which players join and leave them game, as well as some overhead for the synchronization mechanism. As such, in comparison to the other categories of data, player discovery is an extremely low frequency and can use a strict, slow protocol, to ensure players are discovered correctly.

As previously discussed, the requirement of a second round trip renders existing DSPs unsuitable in a MOG context. Thus, a custom sync protocol is required to synchronize the *realtime stream* and *non synced* data categories. However, the player discovery mechanism occurs at a low enough frequency that ChronoSync can be used. Similarly, the static content needs only to be fetched and not synchronized over time meaning a standard Interest / Data exchange can be used.

IV. NDNShooter Sync Protocol

One of the most challenging aspects of building MOGs is the requirement for a high-performance networking solution which is capable of supporting a large number of relatively small packets in a low latency manner. As such, a custom protocol was developed to enable scalable and low latency synchronization of game objects over NDN.

There are two key characteristics of MOG data which can be exploited to provide a more efficient synchronization protocol:

1) Players are only interested in the **newest instance** of a piece of named data. The realtime nature of MOGs mean that players are not interested in historical data for a

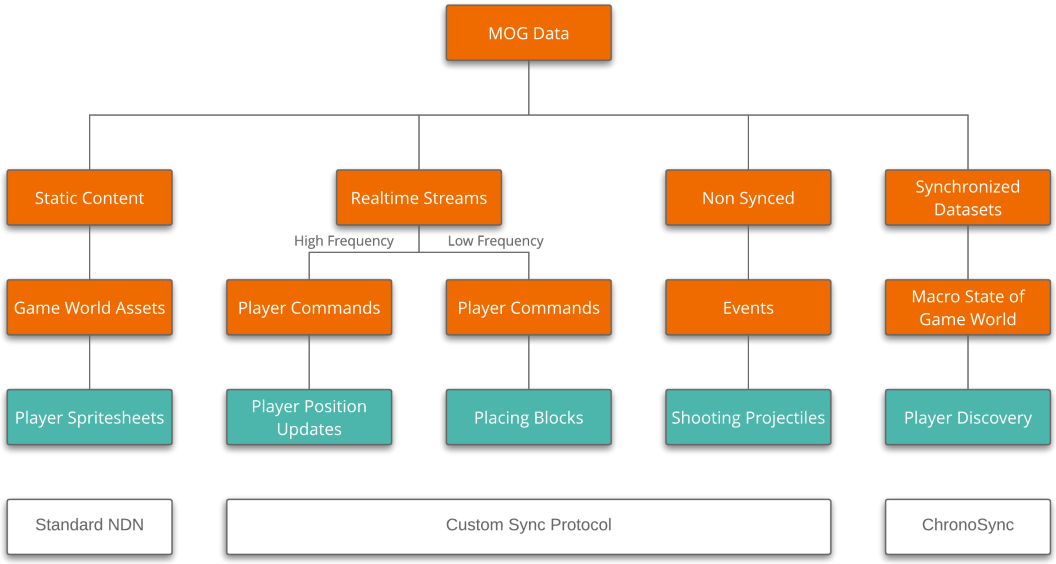

Figure 2. Taxonomy of data found in MOGs (orange), with the corresponding data in NDNShooter (green), and the protocols used to synchronize the data in NDNShooter (white).

game object. This can be exploited by having producers only store and produce the newest instance of their data.

2) Publishers can dynamically control the rate of data production, depending on the state of the game object(s) they are responsible for. For example, a publisher responsible for a game object's position can throttle the rate of updates published if the game object is standing still. This characteristic suggests that an outstanding *SyncInterest* model, similar to what is used by ChronoSync, would allow consumers to express new Interests immediately after receiving remote updates, which producers can satisfy as soon as they have updates to send.

A. Name Schema

One of the most important aspects of designing an application or protocol which uses NDN is the *name schema*. NDN applications should use a naming convention such that consumers can deterministically construct names for data they are interested in. The name schema used for NDNShooter's game object sync protocol is shown in figure 3.

As seen in figure 3, the number of components used in the name depends on the use case. For example, when producers register the prefix with the NFD, they only use the first 5 components (up to the *sync* component), so that they receive the Interests regardless of the version floor (*vf*) or next version floor (*nextVf*). When consumers express Interests for a piece of data, they only use the first 6 components (up to the *vf* component). Finally, when producers respond with Data packets, they use all 7 of the components for naming the Data packet.

Each of the 7 NDN name components are discussed below.

1) gamePrefix: This component is used to target NDNShooter in the global NDN namespace.

2) gameId: This is used to allow for multiple instances of NDNShooter to be run concurrently and in isolation. Players can only see and interact with other players in the same game, as defined by the *gameId*. The *gameId* is chosen upon launching NDNShooter.

3) playerName: This specifies the name of the player which holds the primary copy of the game object in question. This field is discovered through the player discovery mechanism.

4) objectType: This specifies the type of the game object in question. In the current implementation of NDNShooter, this can be *playerStatus* which refers to the status of a player (position, velocity, health etc), *blocks* which refers to the set of active blocks in the game world that were placed by the player or *projectiles* which refers to the projectiles which the player has previously shot.

5) sync: This specifies that this packet is for use with the sync API as opposed to the interaction API.

6) vf: This represents the *version floor*. This specifies the **minimum** version of the corresponding data that can be used to satisfy the Interest. Producers will only respond to the Interest when they have data with a version number greater than or equal to the version floor. This is used to ensure consumers only ever receive data that is newer than what they have already seen.

7) nextVf: This field is added by the producer and represents the **next version floor** that should be used. For example, if a producer satisfies the Interest with version 10 of the corresponding piece of data, the *nextVf* component in the name of the Data packet will be 11. This field is **not** necessarily an incremented copy of the version floor. Depending on network conditions, players can fail to keep up with remote updates and fall behind. For example, a consumer may request version 10 of a piece of data, even though the producer is at version 100 of the data. In this case, the producer will respond with version

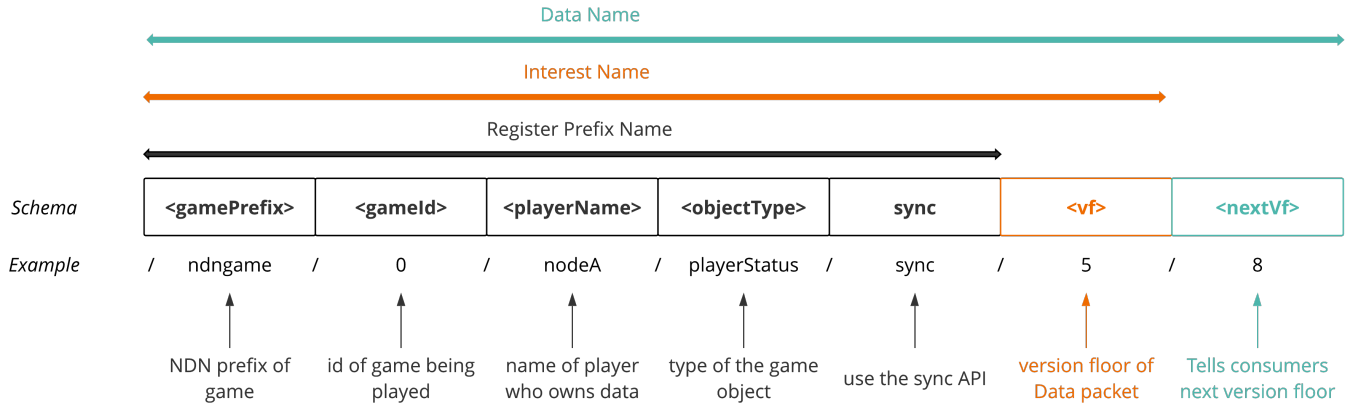

Figure 3. Name schema of NDNShooter's game object sync protocol

100 and set the *nextVf* component to 101. The consumer will extract the *nextVf* component from the name and use it as the *vf* of the next Interest, allowing it to immediately catch up with the producer and to skip all redundant versions.

B. Game Object Sync Protocol in Operation

The operation of the game object sync protocol can be split into three stages - prefix registration, Interest expression and Data production. Assuming the *gameId* is 0, the operation of the protocol for synchronizing nodeA's *PlayerStatus* with nodeB is shown below.

1) Prefix Registration: The first step in the procedure is for nodeA to register the prefix corresponding to nodeA's *PlayerStatus* with its NDN Forwarding Daemon (NFD) [14]. This is done using the *registerPrefix* call provided by the NDN Common Client Library [15]. In this case, nodeA will register the prefix:

$$/ndngame/0/nodeA/playerStatus/sync$$

2) Interest Expression: Assuming nodeB joins the game with *gameId* 0, the player discovery mechanism will discover the other players in this game including nodeA. NodeB will then attempt to fetch the latest version of all of the game objects owned by nodeA, including the *PlayerStatus* of nodeA's avatar. To do this, it will express an Interest for nodeA's *PlayerStatus* using the default initial sequence number of 0. Thus, nodeB expresses an Interest for:

$$/ndngame/0/nodeA/playerStatus/sync/0$$

3) Data Production: Assuming nodeB's Interest gets routed to nodeA appropriately, nodeA will add the Interest into a data structure representing the outstanding Interests for nodeA's *PlayerStatus* that it has not yet satisfied. If the sequence number of nodeA's *PlayerStatus* is less than the sequence number contained in the name of the Interest from nodeB, the Interest will not be satisfied right away and will be deferred until a later time. However, as nodeB requested sequence number 0 of nodeA's *PlayerStatus*, this will certainly be available as all players are given an initial position which corresponds to sequence number 0 of their *PlayerStatus*. Assume nodeA has been in the game for a few minutes and that the sequence number of nodeA's *PlayerStatus* is 90. As 90 is larger than 0 (the version floor contained in the Interest name), nodeA has an updated *PlayerStatus* that has not yet been seen by nodeB. NodeA will create a Data packet which contains the **newest instance** of nodeA's *PlayerStatus*, which is version 90 in this case. Thus, nodeB receives the most up to date version of nodeA's *PlayerStatus*. As nodeB will be receiving the 90th version of nodeA's *PlayerStatus*, the next version floor nodeB should use is version 91 and this is used as the value for *nextVf* in the name of the Data packet produced by nodeA. Thus, nodeA replies with a Data packet of the following form:

$$name : /ndngame/0/nodeA/playerStatus/sync/0/91$$
$$content : version\ 90\ of\ nodeA's\ PlayerStatus$$

C. Benefits of the Sync Protocol

The main benefit of the synchronization protocol is that it does not require separate Interest / Data exchanges for update notifications and update fetching. As previously discussed, latency is one of the biggest factors which negatively impacts the MOG experience. As the protocol only requires a single round trip to fetch remote updates, it is a considerable improvement over using one of the existing dataset synchronization protocols as done in other NDN based MOGs such as Egal Car [11].

Another benefit of the protocol is the that throttling occurs on the producer side. Consumers immediately express new Interests upon receiving Data for a previous Interest, or when an Interest times out. Producers can control the rate at which they satisfy Interests, which in turn has the effect of directly controlling the amount of traffic seen on the network. This is favourable over consumer Interest throttling as producers

have more contextual information about the state of the game object in question. For example, producers can choose not to publish redundant updates when a game object is at rest.

From a consumer point of view, the protocol is extremely simple. Consumers need only maintain a single outstanding Interest for a given piece of data at all times and publishers will respond with the newest version of that data when it is available. This means that all Data received by a consumer is relevant and should be treated as a valid update to the game object. In fact, consumers do not even need to maintain state regarding their version floor, as the next version floor they should use is contained in the Data packet they receive.

As publishers always respond with the latest version of their Data and indicate the next version floor to use, the protocol provides an **automatic catch up** mechanism. This is particularly useful when new players join the game, as it allows them to catch up with existing players within one Interest / Data exchange.

As discussed previously, the three main benefits of NDN in a MOG context are **Interest aggregation**, **native multicast** and **in-network caching**. In order to obtain these benefits, it is imperative that the naming schema used by the protocol contains **no reference to the consumer**.

For example, if the protocol required a producer to be aware of which consumer had expressed a given Interest, the naming schema would have to contain the consumers identity. This would involve including the consumer's *playerName* in the Interest's name. However, as *playerNames* are unique, Interests for the **same** data from **different** consumers would have **different names**, meaning they would not be aggregated at intermediate routers. This in turn would require the producers to produce different Data packets for each consumer, meaning they would gain no benefit from NDN's native multicast. Finally, if Interest names were dependent on consumer identities, there would also be no opportunity for in-network caching.

The game object sync protocol used does **not** require producers to know who expressed the Interest. Similarly, the name schema used makes no reference to consumer names. Thus, the protocol makes use of Interest aggregation, native multicast and in-network caching and these can provide major benefits in terms of network performance.

As every player is interested in the remote updates of every other player, outstanding Interests for the same piece of Data will be aggregated at intermediate routers, largely decreasing the number of packets that must be sent across the network to allow all consumers to receive the remote update. Once newly joining players have caught up using the catch up mechanism, they will begin to express Interests for the same data as the existing players. This represents the steady state of the network and depending on the NDN topology, heavy Interest aggregation can occur in this state.

Similarly, as Interests are aggregated at the producer's local NFD, producers will typically only receive one Interest for a piece of named Data. Once the producer has an update worth publishing, the Data packet generated will be multicasted to all players who are waiting on the remote update. This reduces the amount of work the network module of NDNShooter must perform, as it does not need to separately send the data to each consumer.

Although in the steady state Interest aggregation will be the primary mechanism for reducing network traffic, it is possible that a player with a poor connection could fall behind the Interest aggregation period. For example, if congestion occurs at one of nodeA's upstream links, nodeA may end up requesting data which nodeB has already retrieved. Thus, it is too late for nodeA's Interest to be aggregated. However, if the delay is relatively short, the data can be returned from the CS of the first intermediate router common to nodeA and nodeB. If the delay is longer than the *freshnessPeriod* specified by the producer, the Interest will reach the actual producer of the data. In this case, the catch up mechanism will again enable nodeA to quickly catch up with the other players in the game, allowing nodeA's subsequent Interests to be aggregated once more.

V. NDNShooter Optimizations

A. Consumer Side Position Extrapolation

As discussed previously, *dead reckoning (DR)* is a necessity for a fast-paced game such as NDNShooter. As such, all updates to mobile game objects will contain the game object's velocity vector, as well as the updated position, allowing non-primary copy holders to extrapolate the position of game objects between remote updates.

B. Dead Reckoning Publisher Throttling (DRPT)

The use of DR also enables publishers to throttle their updates based on the state of the game objects they are responsible for. However, as NDN packets contain no information regarding source or destination IP addresses, and the name schema of the sync protocol makes no reference to consumer names, an alternative mechanism is required to enable producers to approximate the positions of their game objects on remote player machines. This can be accomplished using the *version floor (vf)* and *next version floor (nextVf)* components of the sync protocol's naming schema.

As producers tell consumers the *vf* to use on their next Interest (*nextVf*), producers can keep a small cache which uses version floors as keys. Just before a producer sends a Data packet containing an update for a mobile game object, it will write a new entry into the cache using a key k and a value v. Recall that producers append the *nextVf* field to the name contained in the Interest to generate the name for the Data packet. The *nextVf* will be used as the key k when writing to the cache. The content to be contained in the Data packet generated by the producer is a snapshot of the game object that will be sent to the consumers. This snapshot represents the version of the game object that consumers were sent **when they were told to use *nextVf* as the version floor in their next Interest**. The value v to be written to the cache is a composite object containing the snapshot sent to the consumers and the current timestamp. The producer will write the tuple k and v to the cache, and send the Data packet to the consumers.

After some time, the producer will receive a new Interest, with a given version floor, vf_{t2}. However, vf_{t2} will always be equal to the next version floor field of the **previous** Data packet the consumer received, nfv_{t1}. As previously discussed, the next version floor is the key used for writing to the cache. Thus, the producer can use vf_{t2}, which is equal to nfv_{t1} to extract the last snapshot the consumer received and the corresponding time at which it was sent. As the producer knows the extrapolation algorithm in use by the client, it can perform the same extrapolation process using the previous snapshot, previous timestamp and the current timestamp, to obtain an estimate of where the replica copy of the game object would be from the consumer's point of view. The producer will publish an update if:

1) There is no entry corresponding to the version floor in the cache. This indicates that the consumer has fallen considerably behind and has not received a remote update in some time, and that an update should be sent immediately.
2) The primary copy's velocity vector is now different to the velocity vector contained in the previous update. As NDNShooter does not use accelerations, a change in velocity is indicative of a change in direction and thus requires an immediate update.
3) The Euclidean distance between the primary copy's actual position and the estimate of the consumer's extrapolated position is larger than a threshold value T_{dr}.

An important choice here is the threshold value T_{dr}. There is an inherent trade-off between the benefit of network traffic reduction, and the drawback of the inaccuracy introduced in the replica copies, as a result of throttling the publisher update rate. The ideal value for T_{dr} is very dependent on the type of MOG and could also be set dynamically depending on the network conditions at a given time.

C. Interest Management

As discussed previously, *interest management (IM)* is a key component in allowing MOGs to scale. In NDNShooter, players are shown a cropped region of the entire game world, in which their avatar is always fixed at the centre of the viewport. As the player moves their avatar around the game world, the camera moves with it, uncovering new regions in the game world. nAs such, game objects outside of this viewport do not need to be synchronized as thoroughly as those inside the viewport, as they cannot be seen by the player. However, as game objects are mobile, they cannot be disregarded entirely as relative motion between the local player and game objects can cause them to come into the player's viewport.

The IM system in NDNShooter is based only on the distance game objects are away from the local player. A graphical representation of the IM system is shown in figure 4.

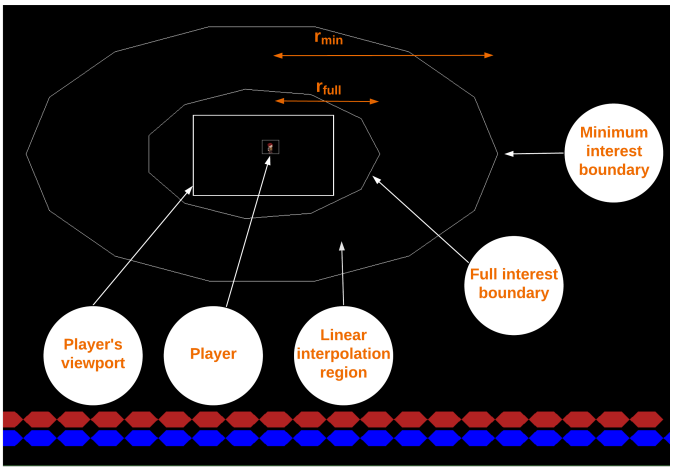

Figure 4. Parameters associated with NDNShooter's IM system

NDNShooter's IM system requires three parameters:

1) MAX_SLEEP*:* This parameter is the *maximum sleep time*. This value represents the maximum amount of time between receiving a Data packet containing a remote update and expressing an Interest for the next Data packet. This should be chosen such that no game object can travel from outside the *minimum interest boundary* (see below) to within the viewport in less than MAX_SLEEP seconds.

2) r_{full}*:* This parameter defines the radius of the *full interest boundary*. This value should be slightly larger than the width or height (depending on which is larger) of the player's viewport. All game objects inside this boundary are considered *fully interesting* and should be synchronized as strictly as possible.

3) r_{min}*:* This parameter defines the radius of the *minimum interest boundary*. This value should be chosen along with MAX_SLEEP to ensure that no game object can travel from outside of the *minimum interest boundary* to within the player's viewport in MAX_SLEEP seconds.

The IM system influences the time between receiving a remote update for a game object and expressing a new Interest to request the next update. The logic here is that the overall number of Interests pushed into the network can be minimized by dynamically controlling how frequently players request updates for game objects, based on how far away they are from the game object.

The IM system calculates the Euclidean distance *d*, between the local player's position and the newly received position for the game object. A *sleepTimeFactor* is then calculated using the following function:

$$sleepTimeFactor = \begin{cases} 0 & d < r_{full} \\ \frac{d - r_{full}}{r_{min} - r_{full}} & r_{full} \leq d \leq r_{min} \\ 1 & d > r_{min} \end{cases} \quad (1)$$

Finally, the actual *sleepTime*, indicating the time to wait before expressing the next Interest is given by:

$$sleepTime = MAX\_SLEEP * sleepTimeFactor \quad (2)$$

The inclusion of an IM system is critical for the scalability of a MOG, and even more so for a massively multiplayer online game (MMOG) as it allows the network traffic to scale as a function of the **density** of players in a certain area, as opposed to the overall number of players.

VI. Testing Methodology

NDNShooter was tested continuously throughout development. This was primarily done using a single machine with multiple instances of the game running as separate processes, each of which communicated via a single NFD. This was an ideal setup for design, development and debugging. However, it is not representative of a real scenario in which multiple players are playing NDNShooter. As such, a more realistic approach was required.

A description of the libraries and frameworks used, and the software developed to facilitate real-world testing of NDNShooter is provided in the following sections.

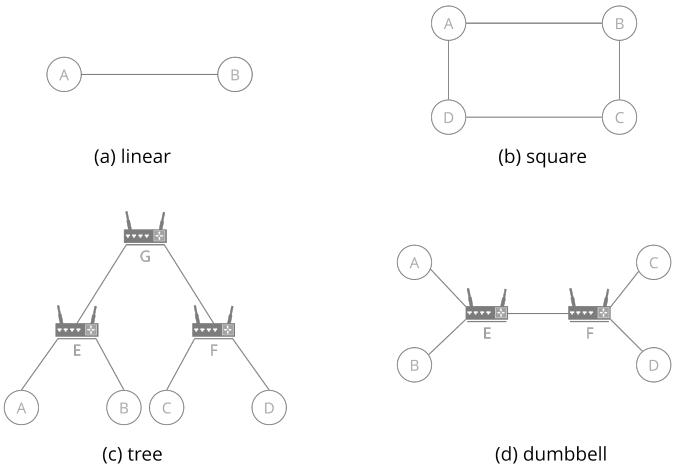

Figure 5. An example of the topologies used for testing NDNShooter

A. Player Automation

In order to enable testing without actual people playing the game, a simple automation mechanism was developed. The automation script used caused players to move in approximate paths, while shooting projectiles approximately once per second and placing blocks approximately once every 10 seconds. A random number generator was used to ensure that a player would not just repeatedly loop through the same moveset. However, a fixed seed was used for the random number generator, providing repeatability across tests using different topologies and parameters, enabling direct comparisons between these tests.

B. Docker

As previously discussed, having several Java processes communicating using a single NFD does not represent a realistic scenario. Thus, an approach using a separate NFD for each player was developed, which closely resembles the real-world use case of multiple players playing on different machines connected by a network.

To accomplish this, Docker was used. A Docker image which contains everything required to run NDNShooter was developed based on Peter Gusev's [16] NDN Docker image. This enabled NDNShooter to be run on any machine which had Docker installed. For each of the topologies tested, a separate *Docker compose file* was created, which contained a *service* for each node in the topology. Thus, each automated game player ran as a *service*, as did each intermediate router.

In order to test the scalability of NDNShooter, dozens of containers needed to be run simultaneously. However, this approach did not scale well on a single machine, and an approach was required to enable deploying the Docker containers across a cluster. Docker also allows for the creation of virtual networks, over which Docker containers can communicate. An overlay network was created, enabling each node in the Docker swarm to communicate. This was done by giving each of the nodes a *network alias*. For example, node A was given the alias *nodea.ndngame.com* and node B was given the alias *nodeb.ndngame.com* etc. Docker Compose supports defining network aliases in the service definition, and this is the reason each node was defined as a separate service.

This provided a mechanism for any two nodes to communicate using a UDP tunnel through the overlay network created by Docker. However, a key component of the testing is defining the NDN topology and seeing how it impacts the performance of NDNShooter. Even though every node can directly access every other node using a UDP tunnel, the actual paths taken by the NDN Interest / Data packets are defined by the FIB of the NFDs of each node.

NLSR [17] is the routing protocol used by NDN to discover nearby nodes (routers) and to build the FIB. The NLSR daemon requires a configuration file which defines a variety of NLSR specific parameters, as well as the name of the node, adjacent nodes and the name prefixes which this node can produce data under.

By creating a configuration file for each node in the NDN topology, and running the NLSR daemon on each of the test nodes with the appropriate configuration file, the NDN topology could be fully defined, and all players will discover the required prefixes and their NFDs will function appropriately.

To examine the impacts of the topology on the performance of NDNShooter, a Python script was developed to perform the following:

1) Allow for topologies to be defined in code, supporting both game players and intermediate routers.
2) Generate appropriate NLSR config files for each node in the topology
3) Generate an appropriate Docker Compose file for the topology, allowing all nodes in the topology to communicate and for the entire test to be runnable with a single command.

The script was then used to generate the required files for testing a variety of topologies such as the ones shown in figure 5.

These topologies were chosen as they offer a good coverage of possible topologies. The *linear* and *square* topologies represent two and four player nodes respectively without any intermediate routers. The *tree* and *dumbbell* topologies model hierarchical topologies, such as what would be found with nodes A and B connected to one Internet service provider (ISP), and nodes C and D connected to another.

VII. Metrics

In order to examine the performance of NDNShooter, several metrics were required. There were two sources of data for these metrics: the NDNShooter processes running on the player nodes and the NFD logs running on the player nodes and intermediate router nodes.

To gather data from the NDNShooter Java processes, Dropwizard's Metrics framework [18] was used. Defining metrics proved to be considerably more challenging than expected due to nature of NDN. The main issue is that Interest packets are aggregated and Data packets are multicasted to consumers. This means there is no way for producers to know which consumer they are serving. Similarly, consumers cannot know if they are obtaining a cached copy of a piece of Data, or if the piece of Data they receive is the result of another player expressing the same Interest at an earlier point in time. Thus, a brief description of the metrics used is given below.

A. Round-Trip-Time (RTT)

One of the main challenges was attempting to understand the temporal performance of the network. Recall that the sync protocol uses a long-lived outstanding Interest model, and that producers only reply when there is an update worth sending. Thus, if a consumer expresses an Interest at time t_{int}, and receives a Data packet at time t_{data}, the RTT can be calculated using equation 3,

$$RTT = t_{data} - t_{int} \tag{3}$$

Consider the case where a consumer calculates a RTT of 500ms. This would suggest that the network is performing very poorly. However, this may be due to the fact that the producer had no updates to send when the Interest arrived, meaning the Interest took considerably longer to satisfy, even though the network may be performing well.

The obvious solution to this problem is to have producers respond with the amount of time that elapsed between receipt of the Interest, and an update becoming available, t_{wait}. This would allow the RTT calculation to be adjusted, yielding the *effective RTT,* RTT_{eff} as shown in equation 4.

$$RTT_{eff} = (t_{data} - t_{int}) - t_{wait} \tag{4}$$

However, due to Interest aggregation, the producer will only see one Interest for a given Data name, regardless of the number of consumers who express the same Interest. Thus, the producer can only measure t_{wait} for the first Interest they receive. Similarly, the Data packet received by all of the consumers will be identical due to the native multicast feature of NDN. Thus, t_{wait} will only be correct for the consumer whose Interest reached the producer first.

This problem does not exist in a typical IP implementation as the producer would receive a request from each consumer and thus be able to calculate t_{wait} accordingly.

For this reason, the first RTT metric RTT was used instead of RTT_{eff} and this effect was taken into account when evaluating RTTs.

B. Interest Aggregation Factor (IAF)

As previously discussed, one of the major benefits of NDN in a P2P MOG context such as NDNShooter is Interest aggregation. The *interest aggregation factor (IAF)* is a measure of how much Interest aggregation is occurring in the network. The IAF of a particular producer is the ratio of the number of Interests seen by the producer to the number of Interests expressed by consumers for Data owned by that producer.

The *IAF* for a player p, iaf_p, can be calculated using equation 5, where P represents the set of all players, n_{seen_p} represents the number of Interests seen by player p and $n_{exp\_x\_p}$ represents the number of Interests expressed by player *x* for Data produced by player *p*.

$$iaf_p = \frac{n_{seen\_p}}{\sum_{x \epsilon P} n_{exp\_x\_p}} \tag{5}$$

C. Cache Hit Rate

As there is no way for consumers to know if they received a cached copy of a Data packet, the easiest way to accurately determine cache rates was to enable debug mode logging in the NFD's *ContentStore* module. This produces log messages indicating whether or not an entry existed in a CS. A Python script was then written to parse the log files of all the participating nodes to determine their cache hit rates.

A more elegant alternative would be to add this functionality to the NFD used by all of the nodes. However, the amount of log messages produced by enabling logging of the *ContentStore* module was sufficiently small that the simpler approach sufficed.

D. Position Deltas

In order to investigate the performance of the dead reckoning (DR) and Interest management (IM) features, a metric was required which captured the error associated with enabling these features and adjusting their parameters.

The *position delta* metric is calculated on receipt of an update for a game object. It is simply the Euclidean distance between the position of the local copy of the remote game object, and the position contained in the received update.

The network benefits of DR and IM can be examined using other metrics, but the more aggressively these systems are used, the worse the gameplay experience becomes. However, as the test players are automated and run in headless mode, the *position delta* was used to attempt to numerically measure the associated impact on gameplay experience.

VIII. Evaluation

In order to validate the design and evaluate the performance of NDNShooter, several tests were performed in a variety of scenarios. As the performance of NDNShooter is heavily dependent on the NDN topology, all tests were repeated using several topologies where appropriate. As updates to the status of players makes up the significant majority of the traffic seen on the network, the evaluation of NDNShooter provided here will only consider the traffic generated from the *PlayerStatus* updates. The results gathered based on the other data types (*blocks* and *projectiles*) were extremely similar to those gathered using the *PlayerStatus*. However, in all tests, **all game mechanics were enabled** but the *block* and *projectile* data is omitted from the evaluation for brevity. All tests were repeated several times and each test instance produced data that was extremely similar. In the interest of clarity, error bars are omitted from the figures as they offer no additional insight.

A. Parameters

The backend implementation of NDNShooter requires several parameters to be defined, many of which can have a substantial impact on performance. Each parameter used in the evaluation is outlined below.

Interest Timeout (s): The length of time in seconds that a consumer will wait for an Interest to be satisfied before retransmitting the same Interest.

Data Freshness Period (ms): The amount of time a Data packet is considered *fresh* after being received by an NFD instance. This is defined on a hop-by-hop basis.

Dead Reckoning (DR) Pub Throttling: This mechanism reduces the amount of updates a publisher will produce by maintaining a view of the game object as seen by the consumers. There are two parameters associated with this mechanism - whether or not it is enabled, and the error threshold which results in an update being produced. The error threshold is a measure of distance, and uses *game world units (GWU)*, where the width of the player's avatar is 1 GWU.

Interest Management (IM): This mechanism reduces the rate at which a consumer will express Interests for a given piece of data based on how far away the game object is from the consumer. This mechanism requires four parameters - whether or not it is enabled, the full interest radius (r_{full}) in *GWU*, the minimum interest radius (r_{min}) in *GWU* and the maximum sleep time MAX_SLEEP in seconds.

Publisher Update Rate (PUR) (Hz): This defines the rate at which publishers will check to see if their game object(s) should be updated. This does **not** directly trigger updates to be sent and instead informs the backend module of whether or not there is an update available. If DR publisher throttling is disabled, the publishers will inform the backend of an update at the defined rate. If it is enabled, the publishers *check* to see if an update is required, at the defined rate.

Publisher Queue Check Rate (PQR) (Hz): As outlined in section IV, when Interests arrive, they are added to the *outstanding interests* data structure. This parameter defines the rate at which publishers will attempt to satisfy these Interests. At the defined rate, the publisher will examine each of the Interests in this data structure, check to see if an update has been published by the game engine and satisfy the Interest if possible.

For all tests the following parameter values were unchanged: $Interest\ Timeout = 1000ms$, $Pub\ Update\ Rate = 30Hz$, $Pub\ Queue\ Check\ Rate = 60Hz$. Finally, each of the tests were allowed to run for approximately 5 minutes, as the results obtained did not appear to change once the test reached the steady state.

B. No Caching, no IM and no DR Publisher Throttling

In order to provide a benchmark for future tests, the simplest form of NDNShooter was tested in which no IM or DR publisher throttling was used and caching was disabled.

Round-Trip-Times (RTT): Each of the four topologies performed similarly in this regard. For this reason, the RTTs are only shown for the *dumbbell* topology. As seen in figure 6, the RTT for the *PlayerStatus* of each of the nodes appears to be approximately normally distributed around the 20 - 40 ms range, with a moderate positive skew. This follows from the *publisher update rate (PUR)* of 30Hz, meaning updates are published every 33ms as no optimizations were used. Thus, the RTT is heavily dependent on the *PUR*, as expected. The RTT can be reduced by increasing the *PUR*, however this will increase the traffic on the network and thus reduce the scalability of NDNShooter.

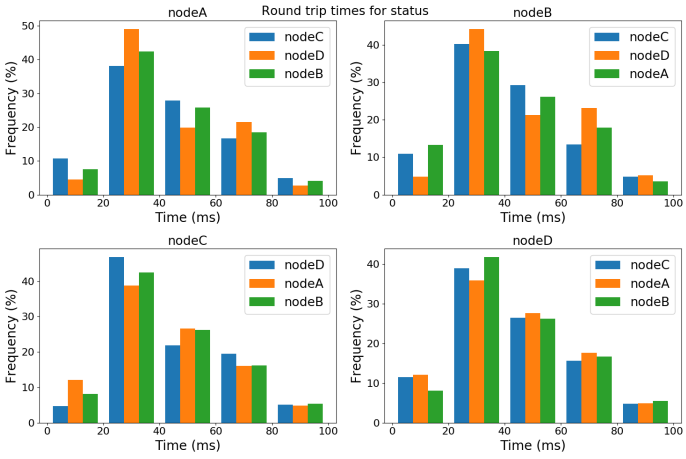

Figure 6. *RTTs* for *PlayerStatus* in the *dumbbell* topology

Position Deltas: As caching, IM and DR publisher throttling are disabled, consumers are receiving a constant stream of the most up to date player positions. This suggests that the *position deltas* should be at their lowest in these tests. However, as the optimizations are disabled, these tests will have the largest amount of traffic on the network. Thus, provided the network is not overwhelmed, the *position deltas* should be extremely low in this case. Otherwise, the lack of network optimizations may affect performance sufficiently that the *position deltas* will be large.

As with the RTTs, each of the topologies performed very similarly in this regard and only the histogram for the *dumbbell* topology is shown.

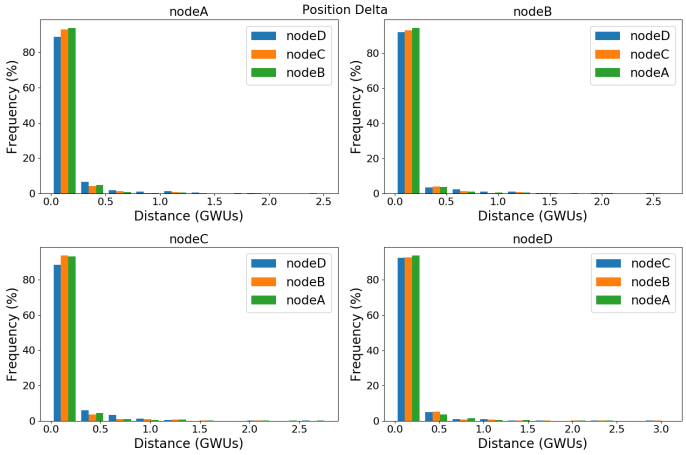

Figure 7. *Position deltas* for players in the *dumbbell* topology. Note the frequencies over *position deltas* over 1.5 GWUs are extremely low, but are non-zero and were left in the histograms to highlight this fact.

The *position deltas* seen in figure 7 show that player positions are synchronized very closely. The figure shows that almost 90% of the time, when a position update was received for a player, the extrapolated position was within half a *GWU* of the new position.

Interest Aggregation Factor (IAF): As previously discussed, one of the main benefits of NDN in a MOG scenario is Interest aggregation. The extent of the Interest aggregation in NDNShooter is shown in figure 8. In these figures, the orange bar represents the total number of Interests expressed for data belonging to a particular node, and the blue bar represents the total number of Interest that node received.

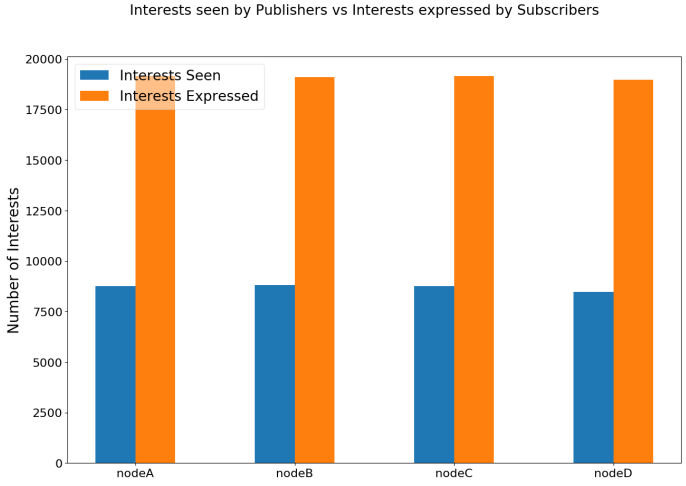

Figure 8. Interest aggregation for *PlayerStatus* in the *dumbbell* topology. The *IAF* of nodes A, B, C and D were calculated to be 0.45, 0.46, 0.45 and 0.44 respectively

As seen in figure 8, a substantial amount of Interest aggregation takes place, with all nodes having an *IAF* below 0.5. This means that all of the nodes had to service less than half of the requests that would otherwise be required in IP based solution.

C. Effects of Enabling Caching

Theoretically, caching can provide benefits to NDNShooter in the case where a node falls behind the Interest aggregation period. However, one of the challenges with caching in NDN is that the *freshness period* (cache lifetime) is defined on a hop-by-hop basis. Thus, *freshness periods* must be kept very low. To examine the impact of enabling caching, the tests were repeated, and the Data packets were given a *freshness period* of 20ms.

By enabling caching, the rate of Interests seen by producers in the *tree* topology decreased by 2 Interests per second for all nodes, as seen in figure 9. The results for the *square* and *dumbbell* were again very similar.

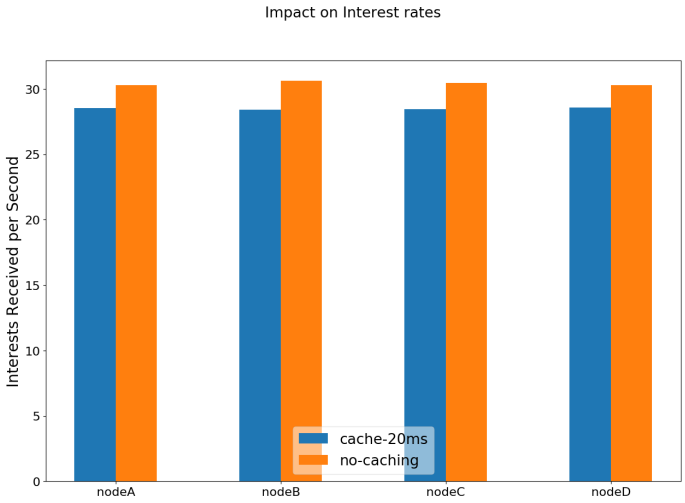

Figure 9. Impact of caching on the Interest rates seen by producers in the *tree* topology

D. Effects of Enabling DR Publisher Throttling (DRPT)

In order to test DR publisher throttling, the tests were again repeated. The 20ms *freshness period* was used once again meaning caching was enabled. The DR publisher throttling threshold was set to 0.5 GWU, which is half the width of the player's avatar. As with the previous tests, only the *dumbbell* is shown, as the *square* and *tree* topologies performed very similarly.

Recall that at a rate of 30Hz (the publisher update rate), the game will check to see if an update needs to be produced. As publisher throttling was enabled for these tests, an update will only be produced if the player's velocity changed (*velocity*), there was no entry in the recent player status cache (*null*), or the DR threshold is exceeded (*threshold*), as discussed in section V. Otherwise, the update is skipped (*skip*). The distribution of the results from the publisher update checks is shown in figure 10 for the *dumbbell* topology.

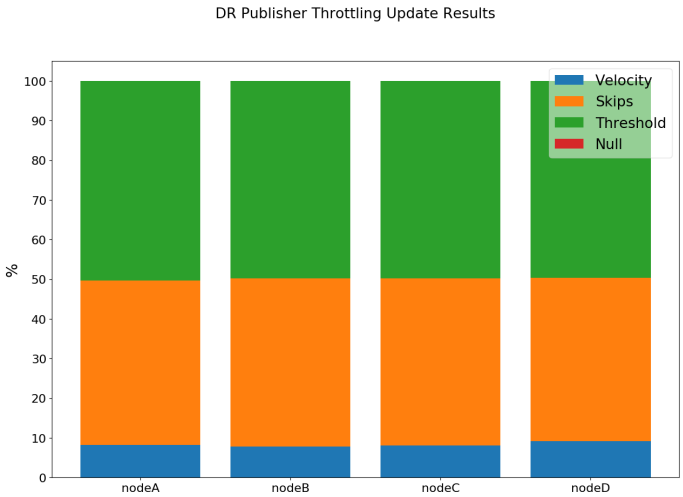

Figure 10. Distribution of results from publisher update checks in the *dumbbell* topology. Note there were no *null* updates required in the test for this topology

As seen in figure 10, **almost half** of the updates can be skipped by enabling DR publisher throttling. This has a substantial impact on the network traffic as player position updates make up the majority of the traffic.

However, as with enabling caching, the impact on the game experience must be considered, and DRPT has the potential to hugely impact the game experience in a negative way, as only half of the player updates are now being published. To examine the impact on the game experience, the *position deltas* were re-examined. Again, the results proved to be very similar across all topologies as they all use the same automation script and thus the results for all but the *dumbbell* topology are omitted.

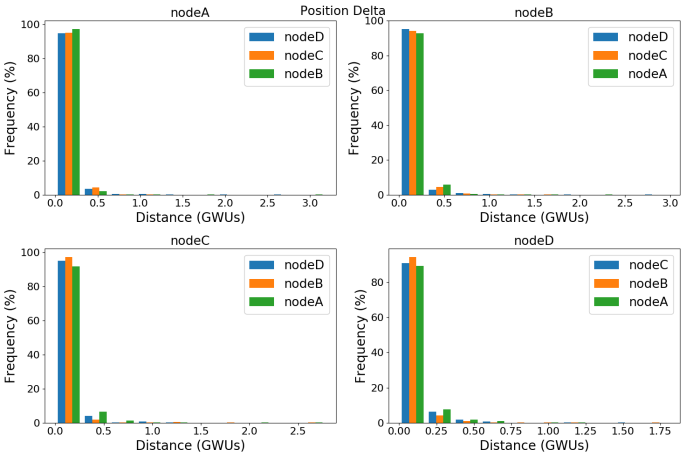

Figure 11. *Position deltas* with DR publisher throttling enabled for *dumbbell* topology.

The *position deltas* shown in figure 11 are very similar to those shown in figure 7, which shows *position deltas* for the benchmark test were all optimizations were disabled. This indicates that there is very little impact on the game experience when DR publisher throttling is enabled with a threshold of 0.5 GWU.

As approximately half of the updates were skipped due to DRPT, the rate of Interests seen by the publishers also dropped accordingly, as seen in figure 12.

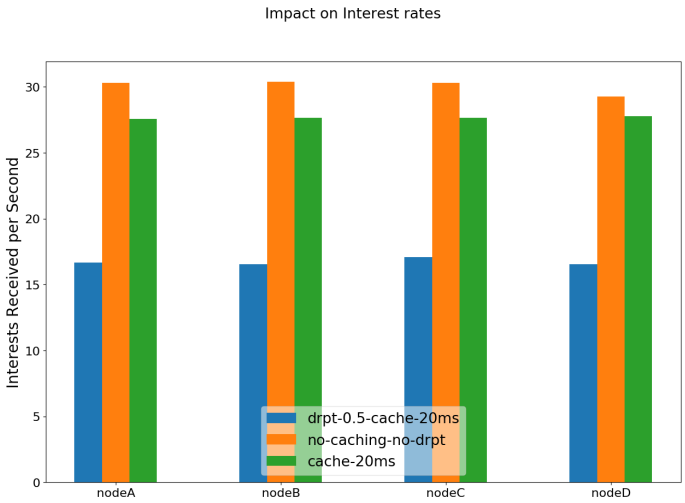

Figure 12. Effects of enabling caching with a *freshness period* of 20ms, and DR publisher throttling (*DRPT*) with a tolerance of 0.5 GWU on the Interest rate seen by nodes in the *dumbbell* topology.

IX. Scalability Testing

In order to test the scalability of NDNShooter, a new topology was built which contained 16 players and four intermediate routers. The topology is shown in figure 13.

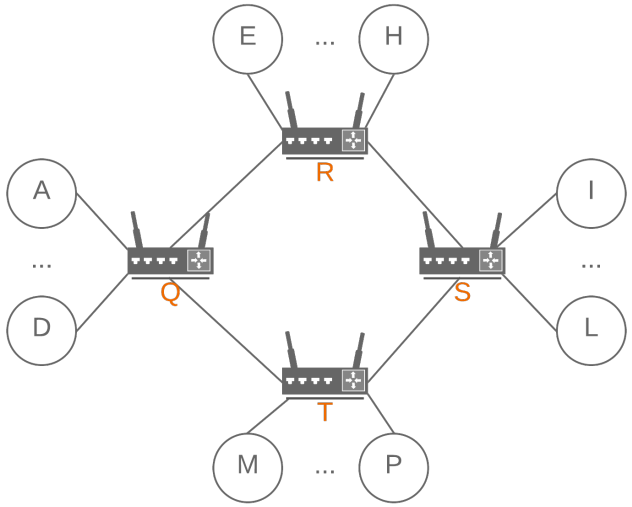

Figure 13. The topology used for testing the scalability of NDNShooter, consisting of 4 intermediate routers (*Q-T*) arranged in a *square* topology, with 4 player nodes behind each router (*A-P*).

A. Benchmark with DRPT and IM Disabled

As before, the test was conducted with some of the optimizations disabled to provide a benchmark. In this case, *DR publisher throttling (DRPT)* and *Interest management (IM)* were disabled, and caching was enabled with a *freshness period* of 20ms as before. The *RTTs* and *Position Deltas* for

each of the 16 players are shown in figure 14 and figure 15 respectively and the effect of Interest aggregation is shown in figure 16.

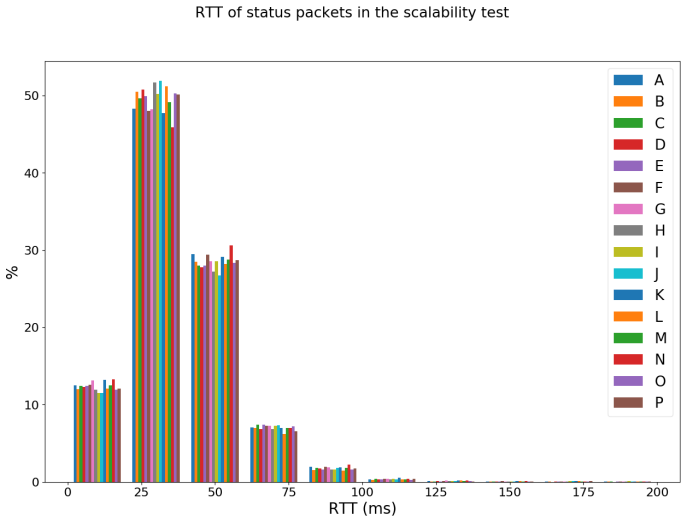

Figure 14. *RTTs* for each of the nodes in the scalability test with no IM or DRPT. As before, there is a peak around 33ms which corresponds to the publisher update frequency of 30Hz.

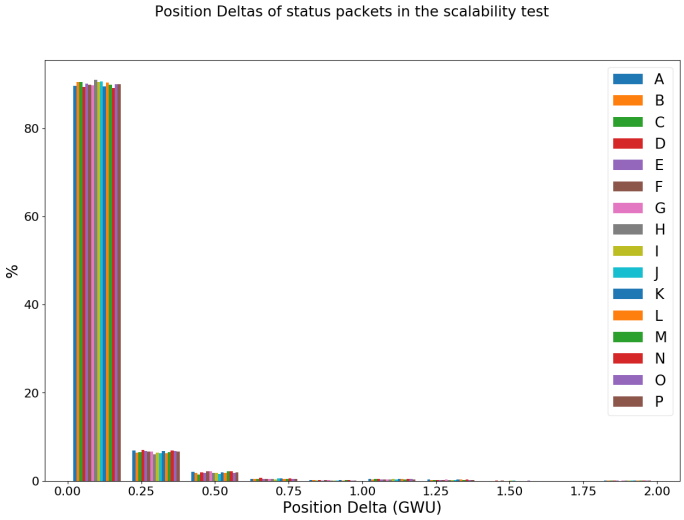

Figure 15. *Position Deltas* for each of the nodes in the scalability test with no IM or DRPT. As before, the values are very close to zero in most cases indicating the players' positions are very tightly synchronized, even with 16 players.

As seen in the figures above, NDNShooter scales up to 16 players very well and the results are very similar to those shown in figure 6 and figure 7, which show the benchmark results for the *dumbbell* topology which only contained four players.

Finally, the IAF of each node in the scalability is shown in figure 16. The IAFs ranged from 0.07-0.09 in the scalability test, which shows that the Interest aggregation mechanisms scales appropriately with the number of players.

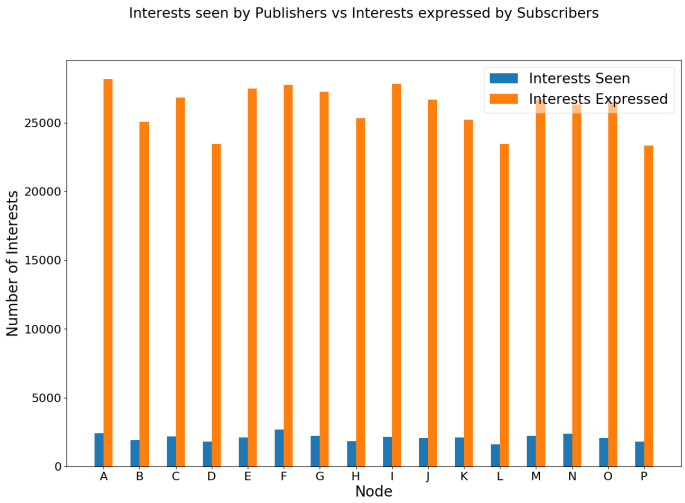

Figure 16. The effect of Interest aggregation in the scalability test with no IM and no DRPT. The *Interest aggregation factor (IAF)* ranged between 0.07 and 0.09 in this case, which is significantly lower than the best *IAF* seen in previous tests were optimizations were enabled. However, *IAFs* are expected to decrease as more players are added, due to the overall larger number of Interests expressed.

B. Enabling DRPT and IM

Finally, the test was repeated three more times, once with only IM enabled, once with only DRPT enabled and once with both enabled and the resulting Interest rates for each of the four scenarios are shown in figure 17.

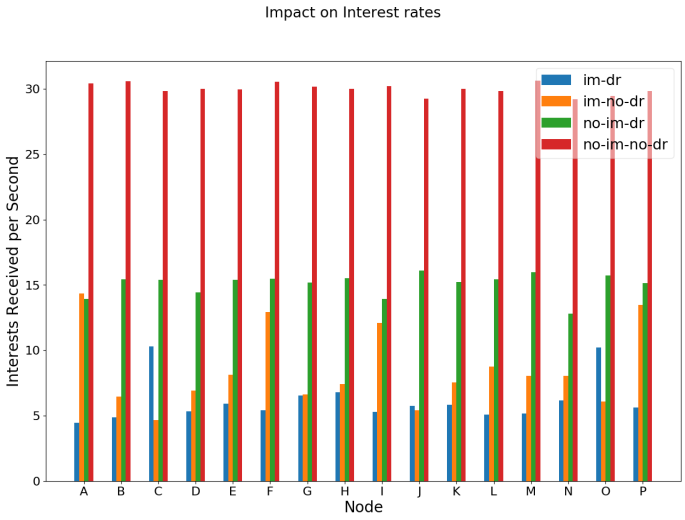

Figure 17. The impacts of enabling IM, DR and both IM and DR on the Interest rates seen by producers in the scalability tests.

Although IM will diminish the effectiveness of DRPT to a certain degree, the two optimizations are entirely compatible. The combined use of DRPT and IM should be the most beneficial in terms of reducing Interest rates and the results shown in figure 17 agree with this theory.

There are a small number of exceptions in which one of the optimizations out performs the combination. An example of this is the difference in Interest rates seen by node C in figure 17 as a result of enabling and disabling the optimizations. IM

alone caused a greater reduction in Interest rate for node C than the combination of IM and DRPT. These discrepancies are most likely a result of small differences in how the game played out. For example, if node C ended up becoming very isolated from other players, IM would cause it to see a much-reduced Interest rate.

X. Conclusion

The performance of NDNShooter suggests that NDN is an excellent candidate for MOG communications. The variety of the data produced by MOGs requires several synchronization strategies for optimum performance. Existing solutions for dataset synchronization such as ChronoSync can be used for the lower frequency data types such as player discovery, though a protocol designed for the specific context will likely be required to synchronize the high frequency data found in MOGs, as was the case in NDNShooter.

NDNShooter provided an ideal test case for the design, implementation and evaluation of various synchronization protocols, which lead to the development of the novel synchronization protocol used in NDNShooter.

The testing performed in section VIII showed the benefits of using NDN as the communication mechanism. The main benefits were *Interest aggregation* and *native multicast*. These factors combined to reduce the traffic seen by the producers by over 50%, and enabling caching reduced the traffic by a further 10% on average. Critically, this substantial reduction in network traffic did not result in any negative impacts on gameplay, as seen by the reasonable values for *RTTs* and *position deltas*.

The DRPT optimization also proved to be extremely effective, enabling publishers to skip between 40%-60% of the *PlayerStatus* updates, which would have otherwise been published to the network. Again, this reduction had no significant impact on the gameplay as seen by the absence of change in the distribution of *position deltas* between the tests with DRPT enabled and disabled.

NDNShooter also appeared to scale very well and supported up to 16 players without any noticeable impacts on the gameplay. Again, this is largely due to the *Interest aggregation* and *native multicast* features offered by NDN. Similarly, the design of the sync protocol used by NDNShooter allows consumers to store all of the state required for synchronization, meaning producers need not store state for each consumer, which provides high scalability with respect to the number of consumers.

The addition of the IM optimization enabled NDNShooter to scale intelligently by taking the state of the game world into account. By using both DRPT and IM in the scalability test with 16 players, producers saw an average of only 5 Interests per second for updates to their *PlayerStatus*, in comparison to an average of 30 when these optimizations were disabled.

A. Future Work

Due to the time limitations associated with this project, several features and research areas were omitted. One of the main goals of the design and implementation of NDNShooter was extensibility, allowing for the addition of more complex game mechanics and networking solutions in the future. In order to continue to investigate using NDN for MOG communications and to further develop NDNShooter, the following work is proposed for the future:

1) IP Based Backend Module: As the backend module used in NDNShooter is entirely modular, an IP based backend module could be developed to facilitate a direct comparison between an IP based solution and the NDN solution developed during this research.

2) Dynamic Freshness Period Setting: As previously discussed, one of the major challenges associated with enabling caching in NDNShooter is the fact that *freshness periods* are defined on hop-by-hop basis. Thus, appropriate values for *freshness period* are very dependent on the topology. A mechanism which attempts to set this value intelligently would be a major step towards increasing the amount of caching possible in NDNShooter.

3) Support NPCs: Adding NPCs to NDNShooter should be a relatively simple endeavour as they are essentially a simpler form of players. Thus, most of the synchronization mechanism that would be required for NPCs has already been implemented. Load balancing NPCs across the players in the game could potentially be done alongside player discovery, by distributing the set of NPCs each player is responsible for along with the player's name. Examining the increase in network traffic due to the addition of NPCs, and comparing it to the increase in network traffic due to a new player joining the game would also be an interesting area of research.

4) Larger Scalability Testing: As the scalability of NDNShooter was unknown prior to the evaluation, the scalability tests were limited to 16 players so that the resulting data could be easily analysed to gain a deeper understanding of how the game scales. As shown in section IX, NDNShooter could easily support 16 players without causing any noticeable impact on game performance. As such, larger scalability tests are required to determine the upper limit to the scalability of NDNShooter.

REFERENCES

[1] Ndn executive summary. https://named-data.net/project/execsummary/. [Online; accessed 11-April-2019].

[2] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael Plass, Nick Briggs, and Rebecca Braynard. Networking named content. *Communications of the ACM*, 55(1):117–124, 2012.

[3] Shang Wentao, Yu Yingdi, Wang Lijing, Afanasyev Alexander, and Zhang Lixia. A survey of distributed dataset synchronization in named data networking. Report, 2017. [Online; accessed 11-April-2019].

[4] Marc Mosko. Ccnx 1.0 collection synchronization. In *Technical Report*. Palo Alto Research Center, Inc., 2014.

[5] isync: A high performance and scalable data synchronization protocol for named data networking. https://named-data.net/publications/poster_isync/, September 2014. [Online; accessed 11-April-2019].

[6] Zhenkai Zhu and Alexander Afanasyev. Let's chronosync: Decentralized dataset state synchronization in named data networking. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2013.

[7] Pedro de-las Heras-Quirós, Eva M Castro, Wentao Shang, Yingdi Yu, Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. The design of roundsync protocol. Technical report, Technical Report NDN-0048, NDN, 2017.

[8] Minsheng Zhang, Vince Lehman, and Lan Wang. Scalable name-based data synchronization for named data networking. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[9] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Computing Surveys*, 46(1):9–9:51, 2013. [Online; accessed 11-April-2019].

[10] Patrick J Walsh, Tomás E Ward, and Séamus C McLoone. A physics-aware dead reckoning technique for entity state updates in distributed interactive applications. 2012.

[11] Zening Qu and Jeff Burke. Egal car: A peer-to-peer car racing game synchronized over named data networking. [Online; accessed 11-April-2019], 2012.

[12] Z. Wang, Z. Qu, and J. Burke. Matryoshka: Design of ndn multiplayer online game. In *ICN 2014 - Proceedings of the 1st International Conference on Information-Centric Networking*, pages 209–210. [Online; accessed 11-April-2019].

[13] Diego G Barros and Marcial P Fernandez. Ndngame: A ndn-based architecture for online games. In *ICN 2015 : The Fourteenth International Conference on Networks*.

[14] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yi Huang, Jerald Paul Abraham, and Steve DiBenedetto. Nfd developer's guide. *Dept. Comput. Sci., Univ. California, Los Angeles, Los Angeles, CA, USA, Tech. Rep. NDN-0021*, 2014.

[15] Ndn common client libraries (ndn-ccl) documentation. https://named-data.net/codebase/platform/ndn-ccl/. [Online; accessed 11-April-2019].

[16] Peter gusev's docker image for an ndn node. https://github.com/peetonn/ndn-docker/tree/master/node. [Online; accessed 11-April-2019].

[17] Vince Lehman, AKM Mahmudul Hoque, Yingdi Yu, Lan Wang, Beichuan Zhang, and Lixia Zhang. A secure link state routing protocol for ndn. In *Technical Report NDN-0037*. NDN, 2016.

[18] Dropwizard metrics homepage. https://metrics.dropwizard.io/4.0.0/. [Online; accessed 11-April-2019].