Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Multiplayer Online Game Communication Using Named Data Networking

Stefano Lupo

A MAI Dissertation submitted in partial fulfillment
of the requirements for the degree of
MAI Computer Engineering

Supervised by Dr Stefan Weber

Submitted to the University of Dublin, Trinity College,
April, 2019

Declaration

I, Stefano Lupo, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed: _____ Date: _____

Summary

This research project focused on designing, implementing and testing a multiplayer online game (MOG) using Named Data Networking (NDN) as the primary communication mechanism. A literature review was conducted resulting in a detailed overview of the current state of the art in the fields of NDN and MOG networking. Modern MOGs were also examined in order to develop a taxonomy of the data generated in MOGs. Each of the categories identified by the taxonomy have different characteristics and require different networking solutions to provide a consistent view of the game world to all players.

The results of this research led to the design and implementation of a 2D MOG in order to facilitate the testing and evaluation of an NDN based backend for the game. The game allows players to move around a virtual game world, shoot projectiles at other players and to place blocks which can be used as cover. These game mechanics were chosen as they provide a good coverage of the categories of data identified by the taxonomy.

A P2P NDN based backend was developed for the game which uses existing NDN software, as well as a novel protocol for synchronizing remote game objects over NDN, to provide a high performance networking solution required by a fast paced MOG.

A system for automating game players was created to facilitate the testing of the game. This was done by creating a Docker container for an automated player and deploying these containers onto an Amazon Web Services cluster using a Docker swarm.

Finally, the performance of the game was evaluated under a variety of network topologies and scenarios by analysing the data produced during the testing period. The results showed that the features of NDN such as interest aggregation and native multicast enabled the game easily support 16 concurrent players. Several networking optimizations were also developed to further reduce the network traffic generated by the game, further increasing scalability.

Abstract

Multiplayer Online Game (MOG) Communication using Named Data Networking (NDN)

Stefano Lupo

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.

Acknowledgements

Thank you to Dr Stefan Weber for the countless hours he spent with me over the course of the project. This research would not have been possible without his understanding of the area, constructive feedback and numerous suggestions.

Contents

List of Figures

List of Tables

List of Code Listings

TODOs

1 Introduction

1.1 Background

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 an new global communication mechanism, maintaining many of they 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*.

1.2 Project Scope

As the project only runs for a limited amount of time, the scope of the research was restricted in order to ensure the most relevant topics could be examined in sufficient depth. As a result, several interesting areas of research were not thoroughly examined such as cheating prevention in MOGs using NDN's security features. Several of the most interesting research areas which fell outside of the scope of the project are proposed as future work in section 9.1.

Similarly, a choice had to be made between examining the feasibility, performance and scalability of NDN in a MOG context, and a direct comparison of NDN and IP in a MOG context. The former was chosen as there would be no benefit in comparing IP and NDN in a MOG context if NDN struggled to support MOGs in the first place. The software developed during this research was designed in a modular way so that the NDN specific code could be easily changed for an IP based module, allowing for the direct comparison between NDN and IP in the future.

2 State of the Art

The first stage of the research focused on a detailed literature review of the two core topics associated with the project - NDN and MOG networking. This section provides an overview of the state of the art of these areas, and concludes with a discussion on closely related projects which represent the intersection of these two fields.

2.1 Named Data Networking (NDN)

As NDN is a relatively new concept, this section provides a detailed description of how NDN works, the benefits it offers, the usage of NDN in a variety of contexts and an overview of the tools and software relating to NDN available today.

2.1.1 Primitives

In NDN, as the name suggests, every piece of data is given a name. The piece of data that a name refers to is entirely arbitrary and could represent a frame of a YouTube video, a message in a chat room, or a command given to a smart home device. Similarly, the meaning behind the names are entirely arbitrary from the point of view of routers. They key aspect is that data can be requested from the network by name, removing the requirement of knowing *where* the data is stored. NDN names consist of a set of "/" delimited values, and the naming scheme used by an application is left up to the application developer. This provides flexibility to developers, allowing them to structure the names for their data in a way which makes sense to the application.

NDN exposes 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. For example, one might request the 100th frame of a video feed of a camera situated in a kitchen by expressing an Interest for the piece of data named */house/kitchen/videofeed/100* and this is done using the Interest primitive.

In the simplest case, the producer of the data, the camera in the kitchen for example,

will receive this Interest packet and can respond by sending the data encapsulated in a Data packet, which will be forwarded back to the requester.

2.1.2 Packet Structures

As outlined in the NDN Packet Specification [3], Interest and Data packets consist of required and optional fields. Optional fields which are not present are interpreted as a predetermined default value. The packet structure for both Interest and Data packets are shown in figure 2.1, where red fields represent required fields and blue fields represent optional fields.

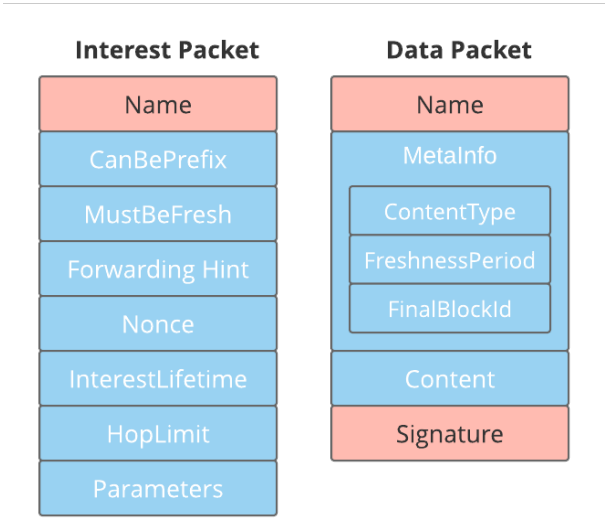

Figure 2.1: The structure of Interest and Data packets [4].

Interest Packet Fields

Name The name of the content the Interest refers to.

CanBePrefix Indicates whether this Interest can be satisfied by a Data packet with a name such that the Interest packet's *Name* field is a prefix of the Data packet's *Name* field. This is useful when consumers do not know the exact name of a Data packet they require. If this field is omitted, the *Name* of the Data packet must **exactly** match the *Name* of the Interest packet.

MustBeFresh Indicates whether this Interest packet can be satisfied by a *content store (CS)* entry whose *FreshnessPeriod* has expired (see section 2.1.3).

ForwardingHint This defines where the packet should be forwarded if there is no corresponding entry in the *forwarding information base (FIB)* (see section 2.1.3). Due to the limited capacity of the FIB, there is a finite number of name prefixes that can be stored. This field typically represents an ISP prefix and is used to tackle the routing scalability issues present in NDN [5][6].

Nonce A randomly generated 4-octet long byte string. The combination of the *Name* and *Nonce* should uniquely identify an Interest packet. This is used to detect looping Interests [3].

InterestLifetime The length of time in milliseconds before the Interest packet times out. This is defined on a hop-by-hop basis, meaning that that an Interest packet will time out at an intermediate node *InterestLifetime* milliseconds after reaching that node.

HopLimit The maximum number of times the Interest may be forwarded.

Parameters Arbitrary NDN implementation data to parameterize an Interest packet.

Data Packet Fields

Name The name of the content the Data packet refers to.

ContentType Defines the type of the content in the packet. This field is an enumeration of four possible values: *BLOB, LINK, KEY* or *NACK*. *LINK* and *NACK* represent NDN implementation packets, while *BLOB* and *KEY* represent actual content packets and cryptographic keys respectively.

FreshnessPeriod This represents the length of time in milliseconds that the Data packet should be considered fresh for. As Data packets are cached in the CS, this field is used to approximately specify how long this packet should be considered the newest content available for the given *Name*. Consumers can use the *MustBeFresh* field of the Interest packets to specify whether they will accept potentially stale cached copies of a piece of Data and the *"staleness"* of the Data is defined using the *FreshnessPeriod* field.

FinalBlockId This is used to identify the ID of the final block of data which has been fragmented.

Content This is an arbitrary sequence of bytes which contains the actual data being transported.

Signature This contains the cryptographic signature of the Data packet.

An important note to make is that neither the Interest nor Data packets contain any source or destination address information. This is a key component of NDN as it allows a single Data packet to be reused by multiple consumers.

2.1.3 Basic Operation

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 an FIB used on IP routers, however NDN supports multipath forwarding (see section 2.1.5), enabling a single Interest to be sent upstream through multiple interfaces.

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. As with all caches, the CS is subject to a replacement policy, which is typically *Least Recently Used (LRU)*.

NDN also uses a *Face* abstraction. An NDN Face is a link over which NDN Interest and Data packets can flow. A Face can represent a physical interface such as a network card, or a logical interface such as an application producing data under a certain namespace.

The operation of an NDN node on receipt of an Interest packet is shown in figure 2.2.

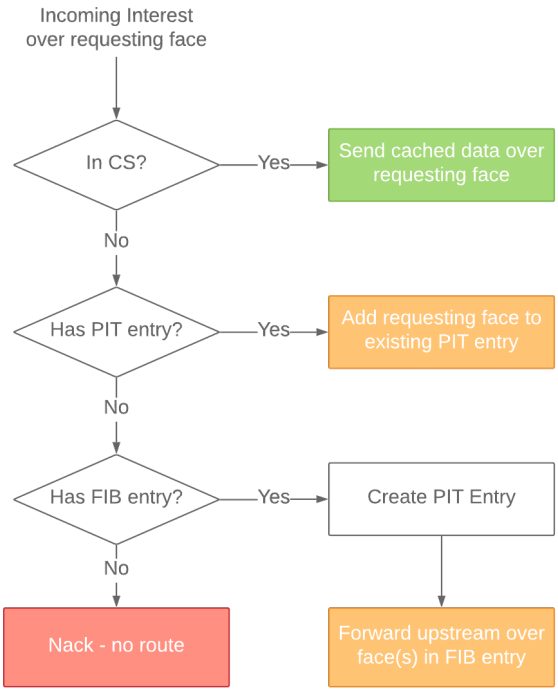

Figure 2.2: Operation performed by an NDN node upon receiving an Interest

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 simply sent back over the requesting Face and the Interest packet is satisfied.

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.

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. In some implementations, the Interest could also be forwarded based on the *ForwardingHint* if one is present. 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.

The operation of an NDN node on receipt of a Data packet is shown in figure 2.3.

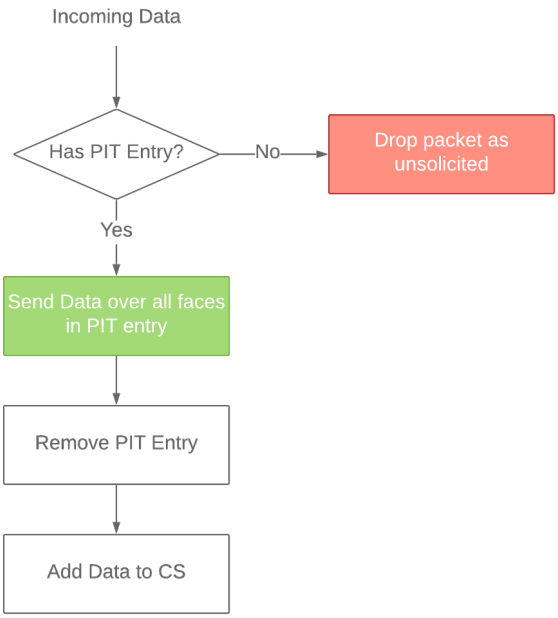

Figure 2.3: NDN operation on receiving Data

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.

2.1.4 Naming

As with IP addresses, NDN names are hierarchical. This can be beneficial to applications as it allows for naming schemes which model relationships between pieces of data. In order to support retrieval of dynamically generated data, NDN names must be deterministically constructable. This means there must be a naming convention agreed upon between producers and consumers, to allow consumers to fetch data [7].

The names of Data packets can be more specific than the names of the Interest packets which solicit them. That is, the Interest name may be a prefix of the returned Data name. For example, a producer of sequenced data may respond to Interests of the form */ndn/test/<sequence-number>*. In this case, the producer would register the prefix */com/test* in order to receive all Interests, regardless of the sequence number requested. However a consumer may not know what the current sequence number is. Thus a convention could be agreed upon such that a consumer can express an interest for */com/test* and the Data packet that will be returned will be named */com/test/<next-producer-*

sequence-number>, allowing the consumer to catch up to the current sequence number. This method is used in the synchronization protocol outlined in section 4.4.

2.1.5 Routing and Forwarding

IP routers use *stateful routing* and a *stateless forwarding plane*. This means that routers maintain some state on where to forward packets given their destination IP addresses (stateful routing). However, when it comes to actually forwarding packets, the packets are sent over the chosen route and forgotten about (stateless forwarding). NDN on the other hand, uses both stateful forwarding and routing [8] in order to accomplish routing packets by name and not address, as seen by the usage of a PIT. NDN routing is done based on the name fields of Interest and Data packets and will select the route which has the *longest prefix* in common with the name in the packet.

As discussed in section 2.1.4, NDN names are hierarchical. This allows NDN routing to scale, in a similar manner to how routing scales by exploiting the hierarchical nature of IP addresses [8].

The use of a stateful forwarding plane is NDN has some drawbacks such as added router operation complexity and the addition of a new attack vector through *router state exhaustion* attacks, due to the limited size of the PIT [9].

However, there are three key benefits offered by NDN's stateful forwarding plane - *multipath forwarding*, *native multicast* and *adaptive forwarding*.

Multipath Forwarding

One of the challenges of routing IP packets using a stateless forwarding plane is ensuring that there are no forwarding loops. Otherwise a single packet could loop endlessly throughout the network. The typical approach to solving this problem is to use the *Spanning Tree Protocol (STP)* [10] to build a loop free topology. This results in a single optimal path between any two nodes in a network and disables all other paths.

However, as NDN uses a stateful forwarding plane, looping Interest packets can be detected and stopped. As discussed in section 2.1.2, Interest's contain a *Nonce* field, allowing Interests to be uniquely identified. If an NDN router sees an Interest which is identical to an Interest in the PIT, the Interest is ignored as a loop has been detected. That is, the usage of a PIT prevents looping. Similarly, as Data packets take the reverse path of the Interest packets, they also cannot loop.

This means NDN can natively support multipath forwarding. This is done by allowing multiple next hops for a given entry in the FIB. This provides flexibility in the routing protocols which can be used with NDN and offers several benefits such as load balancing

across entries in the FIB. Thus, to take advantage of the native multipath forwarding capabilities, a NDN specific routing protocol was developed (see section 2.1.10)

Native Multicast

As discussed in section 2.1.3, when a router receives an Interest which matches an entry in the PIT, it does not forward the second Interest upstream. Instead it adds the Face over which the incoming Interest was received to the PIT entry. Once the data for the Interest reaches the router, it forwards the Data packet to **all** of the faces listed in the PIT entry. Thus, NDN natively supports multicast as a producer may produce a single Data packet and have it reach many consumers.

Adaptive Forwarding

As NDN's forwarding plane is stateful, routers can dynamically adapt where they forward packets as the needs arise. Routers can track performance metrics such as round-trip-times of upstream connections and can use this information to detect temporary link failures, or poorly performing links and route around them.

2.1.6 In-Network Storage

As discussed in section 2.1.3, a Data packet is entirely independent of who requested it or where it was obtained from, allowing a single Data packet to be reused for multiple consumers [4]. The CS of routers provides a mechanism for opportunistic in-network caching, which can help reduce network traffic for popular content.

NDN also supports persistent in-network storage in the form of *repo-ng* [11], which supports typical remote dataset operations such as reading, insertion and deletion [12]. This mechanism provides native network level support for Content Delivery Networks (CDNs) and can allow applications to go offline for longer periods of time while their content is served from in-network repositories [4].

2.1.7 Forwarding Strategies

The choice of how to forward packets in NDN is defined by a *forwarding strategy*. Several strategies have been designed for NDN such as *Best Route*, *Multicast* and *Client Control* [13]. However, *Best Route* and *Multicast* are the most common. To forward packets, a list of possible next-hops is obtained from the FIB for a given Interest. In the *Best Route* strategy, the Interest is forwarded over the best performing Face, ranked by a certain metric such as link cost or round trip time. In the *Multicast* strategy, Interests are forwarded over all Faces which are obtained from the FIB for a given Interest.

As one would expect, Forwarding strategies play a major role in the performance of an application using NDN. For example, the *Multicast* strategy should be used only in scenarios where multicast is beneficial or required, as it can cause a major increase in the number of Interests which must be sent across the network. However, an application's correctness can also be affected by the forwarding strategy [14]. For example, if a *Best Route* strategy is used in a distributed dataset synchronization context, it is possible that only a subset of participants will see published updates and thus *Multicast* must be used in this context.

2.1.8 Security

Security in NDN differs from existing security solutions in that it focuses on securing the data itself at production time, as opposed to securing communication channels between two hosts.

Typically, the main form of secure communication in the Internet today is using the Transport Layer Security (TLS) protocol, along with the Transmission Control Protocol (TCP) over IP (TCP/IP) [15]. TCP/IP is a mechanism for allowing reliable communication between two nodes in a network, by setting up a long standing communication channel between the hosts. In the case of secure communications, TLS is then used to secure the channel.

NDN on the other hand focuses on securing the Data packets produced in response to Interests. As discussed in section 2.1.2, the NDN packet structure specification requires Data packets to contain a *Signature* field. A cryptographic signature is generated using the producers public key, binding the producer's name to the content. [16].

As NDN uses public key cryptography, all applications and nodes must thus have their own set of keys, and a mechanism for determining which keys can legitimately sign which pieces of data. NDN uses three key components in this regard - *NDN Certificates*, *Trust Anchors* and *Trust Policies*.

NDN Certificates bind a user's name to a key and certifies the ownership of this key [16]. Trust Anchors are the certificate authority for a given NDN namespace, and NDN nodes can verify published certificates by backtracking along the trust chain until a Trust Anchor is reached. Finally, Trust Policies are used by applications to define whether or not they will accept certain packets based on naming rules and Trust Anchors.

These security features could likely be leveraged to reduce the possibility of cheating in MOGs. However, due to the limited time associated with this project, research into the security benefits offered by NDN in a MOG context was not conducted.

2.1.9 NDN Forwarding Daemon (NFD)

The NFD is a network forwarder that implements the NDN protocol [17] and provides an implementation for all of the features described in section 2.1.3 such as the CS, PIT and FIB.

As NDN strives to replace IP as the universal network layer, NDN can run over a variety of lower level protocols such as Ethernet, Bluetooth, TCP/IP and UDP/IP. The NFD provides this functionality by abstracting communication to dealing with *Faces*. *Faces* can be backed by a variety of transport mechanisms such as UDP tunnels, TCP tunnels and Unix sockets. This allows applications using the NDN Common Client Libraries (see section 2.1.11) to communicate with the NFD through the Face abstraction.

The API of the NFD provides a means for creating *Faces*, adding *Routes* and specifying *Forwarding Strategies*.

A typical set up of an application using the NFD is shown in figure 2.4. The NFD requires faces to be created before operation. In this case, as part of the set up procedure, node A would create a *Face* towards node B, backed by a UDP tunnel towards node B's IP address. Node A would also create a *Route* towards node B by specifying the prefix node B is responsible for, along with the ID of the Face previously created. In this case the route would map */ndn/nodeB* to face 2. Note that this process is somewhat automated by using the prefix discovery protocol of NLSR (see section 2.1.10). Finally, node A can specify the *Forwarding Strategy*, or use the default of *Best Route*.

As node B is a producer, it only needs to create a *Face* towards the local NFD (face 7 in this case) and inform the NFD that it will be producing data under the prefix */ndn/nodeB*. This is done using the *registerPrefix* call provided by the NDN client library. This will create the a route in node B's NFD which maps */ndn/nodeB* to face 7.

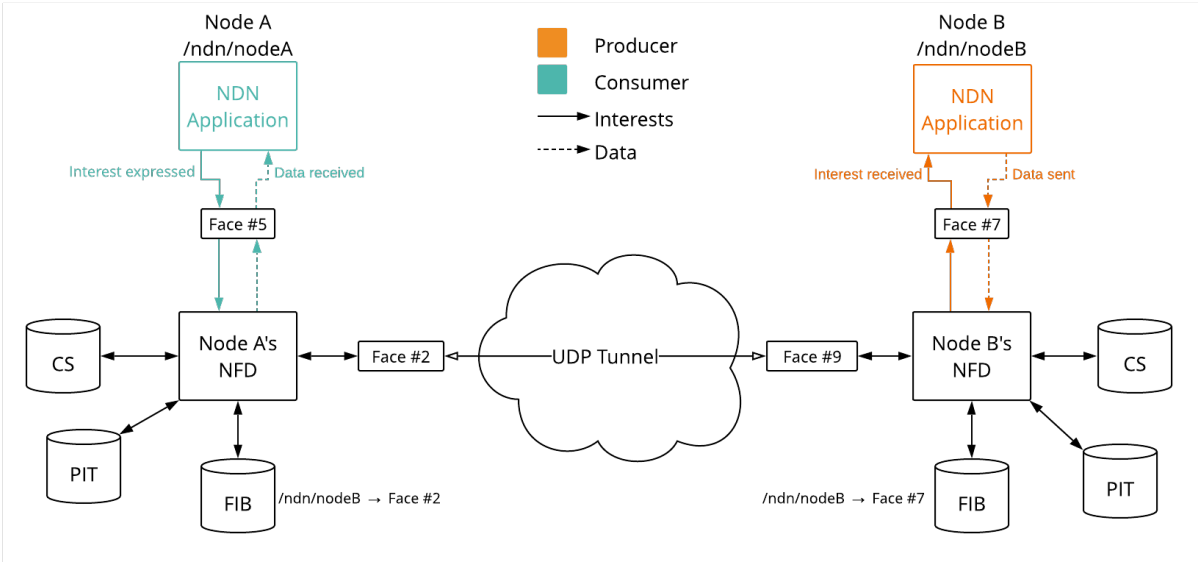

Figure 2.4: An example set up of 2 NFDs for a producer/consumer application.

With the NFDs configured, an example operation would be the following (with some of the basic operations of NDN omitted for brevity):

1. Node A's application creates a face towards the local NFD (face 5 in this case).
2. Node A requests node B's status by expressing an Interest for */ndn/nodeB/status* through face 5.
3. Node A's NFD checks the CS and PIT which are empty and finally determines the next-hop for the Interest is through face 2.
4. Node A's NFD sends the Interest through the UDP tunnel towards node B's NFD which accepts UDP connections on NFD's default port of 6363. This creates face 9 on node B's NFD.
5. Node B's NFD then finds the FIB entry created for */ndn/nodeB* when node B registered the prefix and forwards the Interest over face 7
6. Node B's application will then create the corresponding Data packet and send it over face 7
7. Node B's NFD will check the PIT for a list of faces to forward this Data packet over and will find face 9
8. The Data packet will reach Node A's NFD via the UDP tunnel
9. Node A's NFD will extract the list of downstream faces for this Data packet from the PIT and will send the packet over face 5 to node A's application.

2.1.10 NDN Link State Routing (NLSR)

In order to facilitate router and prefix discovery, the *NDN Link State Routing (NLSR)* protocol was developed. As the name suggests, NLSR is a *Link State Routing (LSR)* protocol [18].

The LSR protocol models the network as a directed, weighted graph in which each router is a node. The main purpose of LSR is to discover the network topology, allowing routers to compute routing tables using a shortest path algorithm such as Djikstra's Algorithm [19][20]. To do this, LSR routers need a mechanism for discovering adjacent routers. However, as this is the process used to *build* routing tables, it cannot make use of existing routing tables. Thus, LSR periodically broadcasts *HELLO* messages over all of the router's interfaces. These messages contain the router's unique address, allowing routers to discover their immediately adjacent neighbours.

The routers then need to reliably disseminate the list of their adjacent neighbours to all other routers in the network, so that all routers have a full view of the network topology. This is done using *Link State Packets (LSPs)*. LSPs contain the list of direct neighbours for a given router, the *edge weight (link cost)* for each of those neighbour connections and a sequence number.

Unlike the *HELLO* messages for neighbour discovery, a router should forward LSPs from a specific router to it's direct neighbours, **once per sequence number**. Thus, routers need to maintain state containing the most recent LSP it has seen for each router. This state is required to determine whether or not a given LSP is newer than what it has already seen, and thus whether or not to forward a given LSP. This information is maintained inside the *Link State Database (LSDB)*. This process is known as the *Flooding Algorithm* and allows all nodes to discover the full network topology and to build their routing tables accordingly.

NLSR is designed as an **intra domain** routing protocol. As it is to be used for NDN, it is imperative that it operates solely using NDN's primitives (see section 2.1.3). Thus, it uses Interest and Data packets as the only form of communication between routers. NLSR differs from the IP based LSR protocol in four main ways [21]:

1. NLSR uses hierarchical naming schemes for routers, keys and routing updates.
2. NLSR uses a hierarchical trust model.
3. NLSR uses ChronoSync to disseminate routing updates (see section 2.1.15).
4. NLSR supports multipath routing.

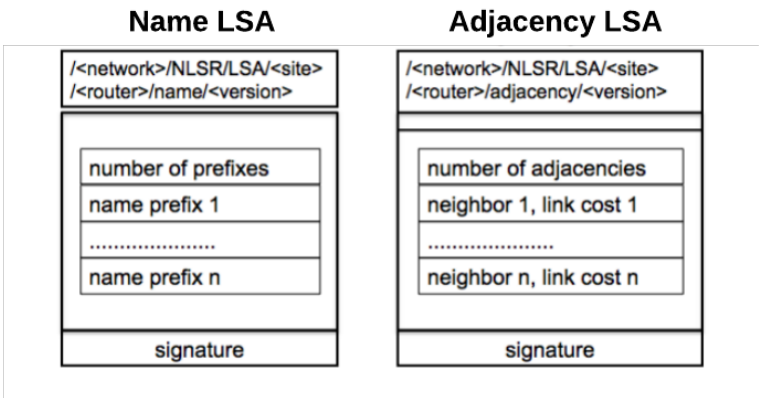

Figure 2.5: Packet structure of Link State Advertisements (LSAs) in NLSR [21] (adapted)

As seen in figure 2.5, NLSR uses *Link State Advertisements (LSAs)* which can be one of two types - *name* or *adjacency*. *Name* LSAs contain the list of prefixes which this router may produce data for, and *adjacency* LSAs contain the list of neighbours a router has.

LSA dissemination is essentially a dataset synchronization problem and NLSR uses NDN's ChronoSync protocol (see section 2.1.15) to synchronize the LSAs. *Name* LSAs can be updated as applications register new prefixes, while *adjacency* LSAs can change as routers go offline and come back online.

As per the ChronoSync protocol, all routers will maintain an outstanding Interest named */<network>/nlsr/sync/<digest>*, where the *digest* is a hash of the LSA dataset from the point of view of the router. In the steady state, all routers will have the same dataset, compute the same *digest* and will express the same Interest. The forwarding strategy of */<network>/com/nlsr/sync/* is set to *multicast* on all routers, meaning routers will forward the Interests over all of the next hops. However, these Interests will still be aggregated at intermediate routers, meaning that in the steady state, there will only be one outstanding Interest over each link in the topology.

If an LSA is changed on a particular router, the router responds to the outstanding sync Interest with a Data packet containing the **name** of the next version of the LSA. Other routers will have their sync Interest satisfied with this Data packet. They can then fetch this updated LSA using a standard Interest packet when convenient, recompute the digest and the steady state will reached once more.

As with LSR, NLSR is responsible for building the FIB and thus the NLSR protocol cannot rely on the FIB when a node first starts. When a new router receives the first response to the LSA sync Interest it expresses, the router may not have an FIB entry

corresponding to the name contained in this Data packet. For this reason, the namespace associated with LSA updates is also set to a forwarding strategy of *multicast*. Thus, the new router will forward this Interest to the it's immediately adjacent neighbours, who will then forward the Interest on the new router's behalf. This process is equivalent to the *Flooding Algorithm* used in LSR.

However, this broadcast should not cause any extra overhead, as Interests will be aggregated by intermediate routers and every router in the network would need to be informed of the LSA change. Thus, Interest aggregation at intermediate routers means NLSR is more efficient at disseminating routing updates than the corresponding *Flooding* algorithm in IP [21].

2.1.11 Tools and Libraries

In order to facilitate the development of NDN applications, the API specified in the NDN Common Client Libraries Documentation [22] has been ported to a variety of popular languages such as C++, Python and Java. Several command line tools under the NDN Tools project [23] have also been developed which provide useful NDN functionality such as pinging remote NFDs, expressing interests, analysing packets on the wire and segmented file transfer.

An NDN simulation tool called ndnSIM [24] has also been developed to facilitate experimentation using the NDN architecture [25].

Another project built by the NDN team is Mini-NDN [26]. Mini-NDN is based on the popular network emulation tool Mininet [27] and allows for the emulation of a full NDN network on a single system. This provides a convenient way to get up and running with NDN and to test NDN applications.

2.1.12 Benefits in a MOG Context

As outlined in section 2.2, MOGs can be built using a variety of architectures, can have a variety of different data types and require extremely high performance networking solutions. As such, MOGs are an excellent test of the performance of new networking technologies and architectures such as NDN. The three major benefits offered by NDN in a MOG context are *Interest aggregation*, *native multicast* and *in-network caching*.

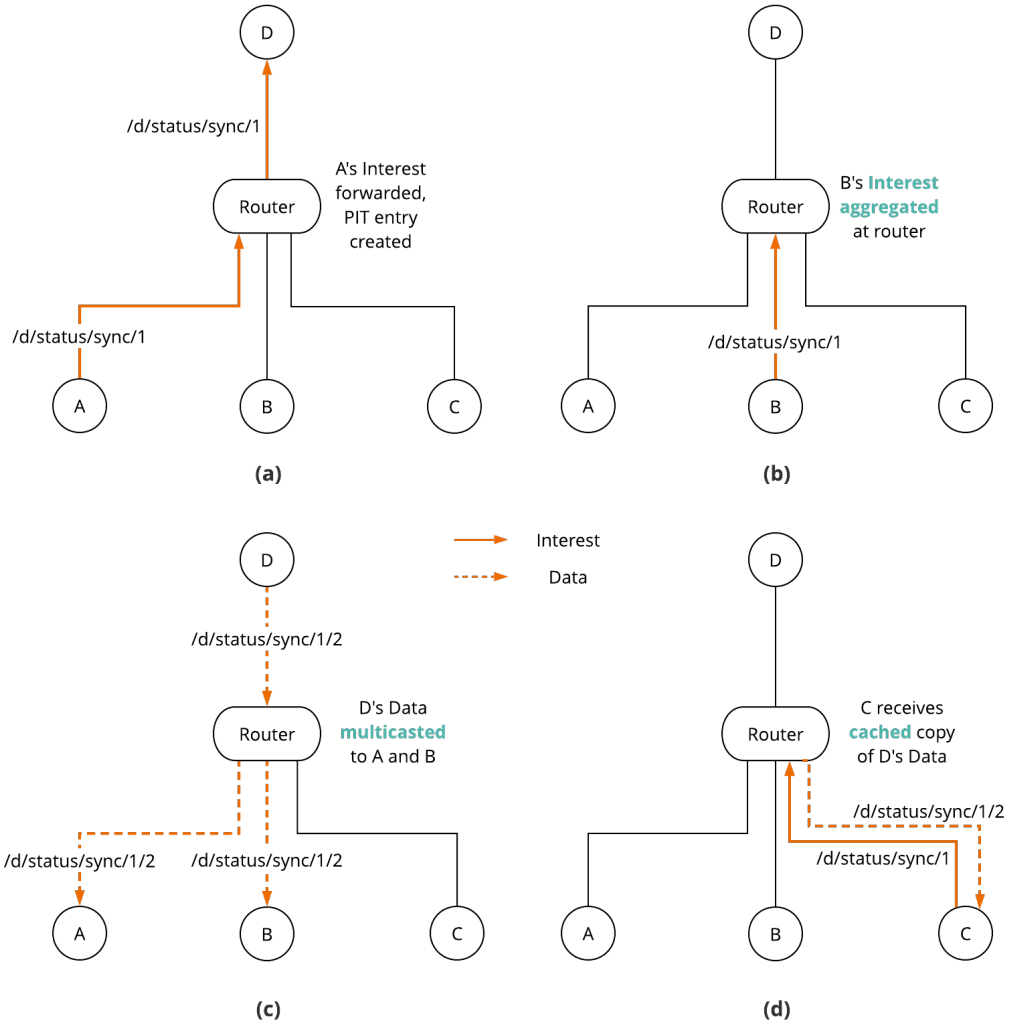

Figure 2.6: Interest aggregation (b), native multicast (c) and in-network caching (d)

Interest Aggregation

As discussed in section 2.1.3, the use of a PIT allows NDN routers to aggregate Interests and has the potential to drastically reduce network traffic in the process. In a P2P MOG architecture, every player is typically interested in the data being produced by every other player. This means there are n^2 logical connections required, where n is the total number of game players, assuming the typical architecture in which every player is responsible for publishing their own updates. In a traditional UDP/IP based implementation, all n^2 of these logical connections are required as actual connections via a UDP tunnels, or something similar.

However, in an NDN based implementation, considering *"connections"* no longer makes sense. Considering the fact that $n-1$ players are likely to be expressing Interests for a given player's Data, Interest aggregation plays a major role in reducing network traffic, as only one instance of the same Interest will be forwarded upstream by each intermediate

router.

As the Interests from separate consumers are forwarded closer and closer to the producer, it becomes more and more likely that they will reach a common intermediate router and be aggregated, although this depends on the topology. The earlier this occurs in the topology the better, as it counteracts the issue stemming from the n^2 logical connections required due to the P2P architecture.

Interest aggregation would also benefit the Client/Server architecture in much the same way, as Interests would be aggregated on route to the server, just as they would be in a P2P architecture while on route to a producer.

Consider the case in which node A expresses an Interest for node D's status (figure 2.6 (a)). This Interest reaches the router, who creates a PIT entry and forwards the Interest to node D. If node B then expresses an Interest for node D's status, the Interest will be aggregated at the intermediate router, as a PIT entry exists for this Data. This means the Interest will not be forwarded upstream to node D a second time, reducing the network traffic seen over the link between the router and node D (figure 2.6 (b)).

Native Multicast

The core concept behind multicast is producing a piece of data once and having it reach multiple consumers. As previously discussed, NDN provides this functionality natively as a direct result of NDN's stateful forwarding plane. Considering MOG networking from a higher level, the architecture is essentially one of *publish-subscribe (pub-sub)*, in which game players (publishers) must publish data to all other players in the game (subscribers). Native multicast is a direct benefit over traditional UDP/IP which requires the same piece of data to be sent to every client, requiring $\mathcal{O}(n)$ sends.

As seen in figure 2.6 *(c)*, once node D has the data to satisfy the Interest expressed by node A, node D will send **one** Data packet back to the router. However, as there are **two** downstream faces in the router's PIT entry for this Data packet, the router will send this Data packet to both nodes A and B. This means node D's Data packet will be multicasted to both nodes A and B.

In-Network Caching

As NDN routers use opportunistic caching via the CS, frequently requested or recently produced Data packets can be cached and served by intermediate routers. This can reduce the round-trip-times of fetching updates from the network and in turn, the overall latency of the MOG.

Although static content (see the taxonomy of MOG data in section 2.2.2) would likely

see relatively high cache rates, the frequency at which static content would be fetched would likely be as low as once per game, meaning the overall network impact would likely be negligible in comparison to the more frequently fetched data which may not be cacheable.

However, considering the outstanding Interest architecture, in which all consumers keep an outstanding Interest and wait for producers to produce the next Data packet (see section 4.4), the effects of caching come into play in the case where a consumer falls slightly behind in fetching remote updates.

For example, if a publisher produces a Data packet every 100ms, it is likely that, in the steady-state, Interests from most of the consumers would be aggregated while the consumers wait on the next packet to be produced. Once this packet is produced, the Data will be multicasted back to all consumers who requested it as previously described.

However, consider the case where a consumer falls slightly behind the other consumers and expresses an Interest for a piece of Data which has already been produced. Without caching, this would require a full round trip all the way to the data producer, putting an extra strain on the network.

Also, while the lagging consumer is fetching the old Data packet, the other consumers will be requesting the **next** Data packet. This means the lagging consumer will continue to remain behind and continue to essentially **double** the rate of Interests seen by the producer, and significantly increase the amount of network traffic required to synchronize a sequenced piece of Data.

However, if caching is used, this Data can be returned from the CS of the first intermediate router who previously forwarded this Interest on behalf of another consumer. Thus, the consumer can potentially receive this somewhat stale Data much quicker and will hopefully catch up with the other consumers. If the consumer continues to lag behind, they will simply continue to obtain cached copies from intermediate routers, limiting the impact on the overall network.

An example of this scenario is seen in figure 2.6 (d), where node C requests the status of node D. However, as this data has already been fetched by nodes A and B, the Data packet will be available in the CS of the router. This will allow node C to receive the Data packet much faster, and reduce the strain on link between the router and node D.

2.1.13 Host Based Applications in NDN

As discussed, NDN's data centric architecture appears to offer several attractive benefits such as in-networking caching and Interest aggregation.

However, most of these benefits only come into play in the case of multiple nodes wishing to consume the same data. Although the switch to a data centric approach makes sense in a variety of modern settings, an interesting research question is to consider how NDN performs for a fundamentally host based application, such as instant messaging or voice communication between two parties. In these scenarios, the benefits of NDN become less clear and the extra complexity associated with using NDN may actually hurt performance.

As outlined in by Van Jacobson et al., *"although data-oriented abstractions provide a good fit to the massive amounts of static content exchanged via the World Wide Web and various P2P overlay networks, it is less clear how well they fit more conversational traffic such as email, e-commerce transactions or VoIP"* [28]. This led to the design and implementation of *Voice over Content-Centric Networks (VoCCN)*, a voice communication protocol capable of running over CCN, analogous to Voice over Internet Protocol (VoIP) [29]. VoCCN conforms to the standards used by VoIP, allowing it to be fully interoperable with VoIP.

One of the main benefits of using NDN in a host based context is the support for *multipath forwarding*. This is particularly useful in VoCCN as voice applications are often used while participants are mobile. Multipath forwarding can be exploited to forward packets towards where a user *might* be located, by taking their mobility into account.

Although the results obtained through testing VoCCN appear to be promising, research into the negative impacts of using NDN for inherently host centric applications is scarce and further examination into the area appears to be required.

2.1.14 Realtime Applications in NDN

Real Time Applications are one of the most challenging types of applications to develop from a networking point of view, typically requiring highly scalable, low latency and high bandwidth communication mechanisms. IP's host-oriented architecture struggles to facilitate applications in which producers must stream real time data to several consumers, as an end-to-end connection between the producer and each consumer is required. These applications are also becoming more and more common with the advent of streaming platforms, such as those offered by television providers.

As discussed by Gusev et al. [30], the shift to the data-centric architecture of NDN provides several key benefits in this context:

Consumer Scalability As NDN provides Interest aggregation and native multicast, the number of consumers that an application can support is a function of the network capacity, as opposed to the producer capacity.

This allows any node, regardless of how small, to produce data to a huge number of consumers, provided the upstream network architecture can support those consumers. An example of this could be a mobile phone device streaming a live event directly to a huge number of people.

Producer Scalability As NDN uses the simple Interest and Data primitives, redundancy and scalability can be accomplished by having multiple producers providing the same data under the namespace. In the event of a producer failure, nothing changes from the consumer's point of view, as the source of the data is always transparent to the consumers in NDN.

Several realtime applications using NDN have been developed. NDN-RTC [31] is a realtime video conferencing library for NDN built on top of WebRTC. Voice over Content Centric networking (VoCCN) [28], as discussed in section 2.1.13, is an NDN equivalent to VoIP. Realtime Data Retrieval (RDR) [32] outlines a protocol for allowing consumers to obtain the most recent frame published by a producer and for pipelining Interests for future frames.

2.1.15 Dataset Synchronization (DS) in NDN

A common requirement in distributed, P2P environments is for nodes to read and write to a shared dataset. An example of a shared dataset is a chat room in which all participants can send messages to all other participants. In order to provide all participants with a common view of the messages sent to the chat room, the underlying dataset must be synchronized by a *dataset synchronization protocol (DSP)*. The importance of DSPs is amplified in a 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.

Centralized approaches require a centralized node which becomes the authoritative source on the state of the dataset. All nodes communicate directly with this node and updates to the dataset are sent through this node. This simplifies the problem considerably at the cost of creating a bottleneck in the system.

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.

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.

NDN can achieve distributed DS by synchronizing the namespace of the shared dataset among a group of distributed nodes [33]. Several protocols have been developed to achieve this including CCNx Sync 1.0 [34], iSync [35], ChronoSync [36], RoundSync [37] and PSync [38]. However, the most relevant protocols at the time of writing are ChronoSync, RoundSync and PSync.

ChronoSync

The primary step in any DS protocol is a mechanism for determining that the dataset has been updated. ChronoSync datasets are organized such that each node's data is maintained separately. Each node has a *name prefix*, representing the name the node's data and a *sequence number*, representing the latest version of that data. The hash of the combination of a node's name prefix and sequence number forms the node's *digest*. Finally, the combination of all nodes' digests forms the *state digest* which succinctly represents the state of the dataset at a snapshot in time. An example of a ChronoSync state digest tree in which Alice, Bob and Ted are interested in the synchronized dataset is shown in figure 2.7.

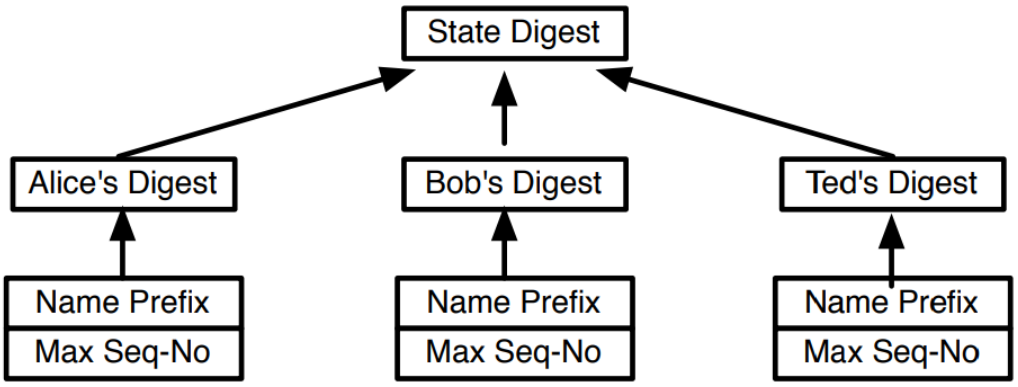

Figure 2.7: ChronoSync digest tree for a dataset synced across Alice, Bob and Ted [36]

Every node interested in the dataset computes a state digest representing the node's current view of the dataset. If all nodes contain the same dataset, all of their state digests will be the same, indicating the dataset is synchronized. ChronoSync uses a *sync prefix* which is a **broadcast** namespace, for example */ndn/chatroom/sync*. All nodes listen for Interests in this namespace. Once a node computes a state digest, it expresses an interest for */<sync prefix>/<state digest>*, for example */ndn/chatroom/sync/a73e6cb*. These are known as *SyncInterests*. Thus, when the dataset is synchronized, all nodes express the **same** SyncInterest.

When a node locally inserts a new piece of data into the dataset, the node recomputes the state digest, which will now be different to the previous state digest. At this point, the ChronoSync library will satisfy the outstanding SyncInterest using a *SyncReply*, which is a standard NDN Data packet. The Data packet used to satisfy the SyncInterest contains the **name** of the data which has been updated as the **content**. The name of the Data packet will simply be the name of the Interest it satisfies.

For example, consider the case in which the current outstanding SyncInterest is */ndn/chatroom/sync/a73e6cb* and Alice's latest sequence number is 5. If Alice inserts a new piece of data into the dataset, Alice will satisfy the SyncInterest with a SyncReply packet named */ndn/chatroom/sync/a73e6cb* which contains */ndn/chatroom/alice/6* as the content. Alice will then recompute her state digest and express a new SyncInterest.

All nodes will receive this SyncReply packet and have the **option** to express a standard NDN Interest to fetch Alice's new data. The other nodes will also recompute the state digest using Alice's new sequence number, and express a new SyncInterest, which will match the SyncInterest expressed by Alice, returning the system to the steady state.

The ChronoSync protocol exploits the Interest aggregation mechanism provided by NDN, meaning that when the dataset is synchronized, there will only be at most one outstanding

SyncInterest on each link in the network.

As a single Interest can only return a single Data packet in NDN, if two nodes produce two different SyncReplies for the same SyncInterest, only one SyncReply will reach a given node. To overcome this, ChronoSync re-expresses the same SyncInterest on receipt of a SyncReply. The second SyncInterest uses an *exclude filter* set to the hash of the content in the SyncReply. This means the same SyncReply will **not** be returned for the second SyncInterest and the second SyncReply can be obtained. This repeats until a subsequent SyncInterest incurs a timeout. ChronoSync also contains features for reconciliation in the event of network partitioning.

The ChronoSync protocol is designed for synchronized write access and must undergo a reconciliation process in the case of concurrent writes to the dataset. It also requires two round trips to obtain the actual updated data - one for SyncReplies and one for fetching the updated data. This limits the effectiveness of the protocol in cases where latency is critical, such as in MOGs.

RoundSync

RoundSync was developed to address the shortcomings of ChronoSync, namely the issues which arise when sync states diverge due to simultaneous data generation. As previously discussed, ChronoSync requires an expensive state reconciliation process when sync states diverge. The shortcoming of ChronoSync was determined to be the fact that ChronoSync uses a SyncInterest to serve two different purposes: (1) SyncInterests enable a node to retrieve updates as soon as they are produced by any other nodes, and (2) it lets each node detect whether its knowledge about the shared dataset conflicts with anyone else in the sync group [37].

RoundSync uses a monotonically increasing round number and limits the number of times a node can produce an update to once per *round*. The key aspect here is that data synchronization is **independent** for each round. This means nodes can continue to publish and receive further updates, while trying to reconcile issues which occurred in previous rounds. RoundSync accomplishes this by splitting up ChronoSync's SyncInterest into a *Data Interest* which is used for fetching updates generated by a node, and RoundSync's own *Sync Interest* which is used solely for detecting inconsistent states within a round [37].

Although RoundSync appears to offer several benefits over ChronoSync, the only available implementation of the protocol is for use with ndnSIM **only** [39].

PSync

PSync was developed as a protocol to allow consumers to subscribe to a subset of a large dataset being published by a producer. Data generated by producers is organized into *data streams*, which are sets of data which have the same name but different sequence numbers. Consumers can subscribe to certain data streams of a producer by maintaining a *subscription list*.

Two accomplish this efficiently, PSync uses two key data structures - a *Bloom Filter (BF)* and an *Invertible Bloom Filter (IBF)*.

BFs are memory efficient probabilistic data structures which can rapidly determine if an element is **not** present in a set. However, BFs can not say for certain that an element is present in a set. BFs use several hash functions to hash the element of interest, resulting in a list of indices into a bit array (one for each hash function).

To insert into a BF, the bits at the corresponding indices provided by hashing the element with each of the hash functions are all set to 1. To determine if an element is **not** in the set, the incoming element is hashed using each of the hash functions, again producing a list of indices. If any of the bits in the array at the list of indices are 0, the element is definitely not in the set, otherwise the element *may* be in the set.

IBFs are an extension to standard BFs which allow elements to be inserted and deleted from the IBF. Elements can also be *retrieved* from the IBF, but the retrieval may fail, depending on the state of the IBF. The operation of an IBF is outlined in appendix A. IBFs also support a set difference operation, allowing for the determination of elements in one set but not in another.

PSync uses BFs to store the *subscription list* of subscribers and IBFs to maintain producers' latest datasets, known as the *producer state*. The producer state represents the latest dataset of a producer and contains a single data name for each of the producer's data streams. These data names contain the data stream's name prefix and the latest sequence number of that data.

Producers in PSync maintain **no state** regarding their consumers and instead store a single IBF for all consumers, providing scalability under a large number of consumers [38]. Consumers express long standing *SyncInterests* which contain an encoded copy of the BF representing their subscription list and an encoded copy of an IBF representing the last producer state they received. The producer can determine if any new data names have been produced by subtracting the current producer state from the producer state contained in the SyncInterest, using set difference operator for IBFs. The producer can then determine whether or not the consumer is actually subscribed to any of these data names using the provided subscriber list. Finally, the producer will either send back the

new data names through a *SyncReply*, or if there is no new data, store the Interest until new data is generated.

Consumers receiving the *SyncReply* can then fetch the new data using standard NDN Interests and update their latest producer state accordingly.

2.2 Multiplayer Online Games (MOGs)

MOGs have become an immensely popular pastime over the past few decades. The games themselves have become increasingly complex and realistic, and there has been numerous advances made which enable modern MOGs to support millions of players. This section outlines the state of the art of several video game related fields such as game development, MOG networking and MOG scaling.

2.2.1 Game Development

Reference games

A huge variety of video game engines, game development frameworks and libraries exist today. The most well known engines and frameworks are those which have been used to make extremely popular games. Valve Software's Source Engine [40] has been used in a number of immensely successful games such as Half-Life, Team Fortress 2, Portal 2 and Counter Strike: Source. Another successful game development platform is Unity [41], which has been used to develop major titles such as Kerbal Space Program and Hearthstone: Heroes of Warcraft.

All of the above engines and frameworks are designed to build extremely detailed games such as the ones listed. However, an emerging sub-industry is that of *independent (indie)* games. As the main area of interest in this project was video game networking, a simpler style of game engine was favoured. Indie games represent a movement away from monolithic game production studios, with huge development teams and budgets, towards developing smaller games, typically with unique art styles and mechanics, which target a niche in the video game market. As such, a large number of smaller scale game engines and frameworks have been developed, one of which is LibGDX [42]. LibGDX is a cross platform, open source game development framework written in Java. It provides an easy to use API which in turn makes use of OpenGL for actual rendering.

2.2.2 Data Taxonomy

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.

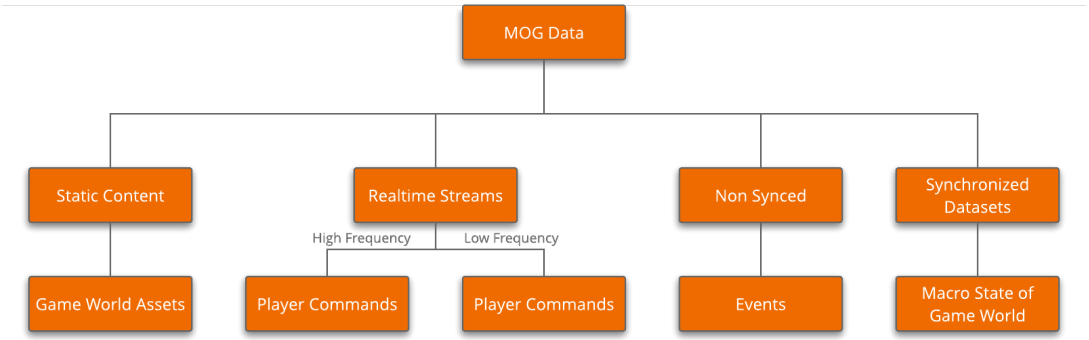

Figure 2.8: Taxonomy of the data found in MOGs

The overall taxonomy of MOG data is shown in figure 2.8 and explained in further detail below.

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. In a simple 2D game, textures are usually stored in *sprite sheets*. Sprite sheets are single images which contain a variety of textures. In order to render a texture, a sub region of the sprite sheet is selected by the game renderer and the pixels within that subregion are drawn to the screen. The reason for using a single sprite sheet which contains a large number independent textures, over a separate file for each texture is performance. Copying a file into the memory of a GPU is a relatively expensive operation in comparison to drawing the texture. Thus, by having multiple textures in a single file, this expensive transfer operation need only occur once and the required textures can be drawn by selecting sub regions of the larger sprite sheet in GPU memory. Static content is typically shipped with the game and read from a file when required. However, 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.

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.

In a FPS style game, players tend to be moving around the game world more often than not. The fluidity of player movement is highly dependent on how quickly player position updates can reach other players. In fast paced games such as FPS games, the player position updates would ideally be sent as frequently as possible. Thus, a good example of a high frequency realtime data stream found in MOGs is player position updates.

However, in the vast majority of MOGs, players can do more than just move around. For example, players may be able to interact with the game world and place blocks at certain positions. Although these player commands still happen relatively frequently, perhaps on the order of a few seconds between successive commands, they are still considered 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. For example, a player may choose to reload their weapon at a certain point in time, which should trigger a reload animation. There is no requirement to synchronize this data over time. Instead, the player simply announces to the network that they are reloading their weapon by publishing an immutable, short lived event.

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.

2.2.3 Architectures

One of the first decisions to make when designing the backend of a MOG is which architecture to use. 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. As one would expect, there are substantial benefits and drawbacks to all of these architectures and the choice of which to use will play a major role on the scalability, consistency, security, ease of development and cost of running of the MOG.

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 [43].

Client/Server (C/S)

C/S is the most common form of MOG architecture today. In the simplest form, C/S consists of a single server which all game players communicate with. The server is the single authoritative source of truth for the game state and holds **all** primary copies. All updates to the game world and game objects occur on the server and these updates are then pushed to all connected players (clients) by the server.

The benefits and limitations of a C/S architecture in a MOG context compared to a distributed alternative are very similar to those found elsewhere in computer science.

The main benefits are the reduced complexity associated with performing all updates in one place and the added difficulty for players to cheat, since the server can determine whether updates are valid prior to performing them. C/S architectures are an ideal choice for games with a small number of players, or which do not require extremely high performance networking solutions such as *real time strategy RTS* games.

The main limitation associated with the C/S architecture is scalability. Modern MOGs require support for hundreds or even thousands of players in a particular game world, and

a single server becomes a severe bottleneck at this scale, regardless of the hardware used. Another potential issue is fault tolerance. Server failures do occur, and the standard C/S architecture provides no fault tolerance whatsoever, meaning the game is entirely unplayable in the event of a server failure.

However, substantial research and engineering has allowed C/S based architectures to meet the demands of modern MOGs and it is still the most commonly used architecture today [43]. The main mechanism for achieving the scalability required is to distribute players among several servers. Clients can be distributed among servers based on their physical locations in the real world or their virtual locations in the virtual world [44].

Distributing players based on their virtual locations is the ideal choice, as it does not entirely segregate players. However, it is more challenging in that a hand-off mechanism is likely required as players cross server boundaries. It can also face scalability issues as players tend to congregate at certain places in the virtual world, such as towns or cities *(flocking behaviour)*, meaning a single server may still struggle due to the density of players in a particular region.

Peer-to-Peer (P2P)

P2P architectures contain no centralized server. Instead, each peer in the network becomes the authoritative source of certain game objects and holds their primary copies. As before, updates are performed only on primary copies. Thus, peers become responsible for accepting update requests, performing updates and disseminating updates to all other peers in the network.

A common method for building MOGs using a P2P architecture is to create an overlay network, backed by a *distributed hash table* [45]. These typically use Pastry to build a *"decentralized, self-organizing and fault-tolerant overlay network, capable of routing messages to other peers in* $\mathcal{O}(\log n)$ *forwarding steps"* [46] and Scribe [47] which provides an application-level multicast infrastructure using the overlay network built with Pastry.

In principle, P2P architectures have the highest potential of all architectures for scalability as every peer that joins the game adds new resources to the system. All of the work is distributed amongst the players in the game, mitigating the requirement for expensive, high performance, centralized servers and providing excellent fault tolerance.

However, building MOGs on a P2P architecture is considerably more challenging than in the C/S architecture. The main issue is that updates to game objects are performed across multiple different nodes at different times. This requires much more complex protocols to ensure a consistent view of the game world is provided to all connected players. Finally,

as players are responsible for accepting, rejecting and performing updates on primary copies, P2P based architectures are much more vulnerable to cheating.

2.2.4 Dead Reckoning

In video games, the rate at which the local game world is updated and redrawn at is known as the *frame rate*. The frame rate of a video game is measured in *frames per second (fps)*. Most modern video games aim to run at a minimum frame rate of 30 fps, while targeting higher frame rates such as 60 or even 100 fps. In the case of a 30 fps frame rate, this would require an update to be available for **all** remote game objects every 33.33 milliseconds (ms). 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 player 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 [48], *"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.

DR Convergence Algorithms

Upon receipt of a remote update packet, it is likely that the extrapolated position does not exactly match the updated position. As such, a DR *convergence algorithm* is required. The most basic form of this algorithm is to directly overwrite the game object's extrapolated position with the new position. However, this can result in remote game objects appearing to suddenly jump to the new updated position, instead of smoothly moving towards it.

The main challenge with DR convergence algorithms is that the difference between the previously extrapolated position and the actual updated position must be reconciled,

while continuing to extrapolate the game object towards a future position.

An example DR convergence algorithm designed by Delaney et al. [49] is given below:

- Let the current extrapolated position of a remote player be p_{ex}
- A remote update is received containing p_{new} and v_{new} representing the player's new position and velocity respectively
- The extrapolated position for remote player p'_{ex} is built using p_{new} and v_{new} which represents the best approximation for where the player will be in n time steps,
- A trajectory is built such that the player will reach p'_{ex} from p_{ex} in n time steps, and the player is moved along this trajectory.

This algorithm allows the extrapolated position to be reconciled with the updated position, while still extrapolating the player towards an approximated future position.

DR Update Throttling

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.

The holder of the primary copy can then use a configurable *threshold value* to determine when an update is required. In the simplest form, this is done by periodically calculating the Euclidean distance between the extrapolated position and the actual primary copy position and publishing an update once this distance exceeds the threshold value.

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.

An interesting component of this method is choosing the threshold value. This value is chosen to minimize network traffic without negatively impacting the apparent consistency of the game world. This choice is similar to choosing the amount of compression to apply to an audio or video stream. Research conducted by Kenny, McLoone, Ward and Delaney examined the impacts of different threshold values by performing experiments with real people, in an attempt to determine an optimal value [50].

2.2.5 Interest Management

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.

However, a somewhat assumed form of IM in MOGs is *temporal* in nature. MOGs are essentially realtime applications, and players typically do not need to know about data that was generated earlier, as the game world has moved on from that point.

There are also opportunities to employ IM by exploiting certain features of the actual game. For example, in a team-based shooting game, the position and status of allies could likely be synchronized less aggressively than that of enemies, as the player's focus is more likely on the enemies they are fighting.

An important aspect of IM is that there is a computational cost associated with determining what subset of game objects a given player is interested in. Thus, the benefits gained from employing IM must be carefully weighted against the associated computational overhead [51]. In this regard, IM mechanisms which push the computation to the actual players are favoured over those which must be performed server-side. Similarly, trading off the computational overhead for memory overhead, through pre computation ahead of time, can also be beneficial, provided sufficient memory is available.

The simplest form of IM occurs in the case of a top-down style game, where a player's viewport is a cropped view of the entire game world. Game objects outside of the player's viewport can be disregarded as they cannot be seen by the player. However, dynamic game objects cannot be disregarded entirely, as their movement may cause them to enter the player's viewport. Instead, the rate at which updates are published to the player for that game object can be slowed, and the contents of the updates can be reduced to only contain the position and velocity of the game object.

An example of a more complex IM mechanism is *tile based IM*. Tile based IM divides

the entire game world into tiles, such that the physical position of all players and game objects will always be on a tile. If an obstacle blocks a portion of a players view of the world, the player should not be interested in any of the game objects behind that obstacle. The simple distance based form of IM does not handle this case and results in players being interested in more game objects than necessary. Tile based IM solves this problem by taking the actual game world into account [52].

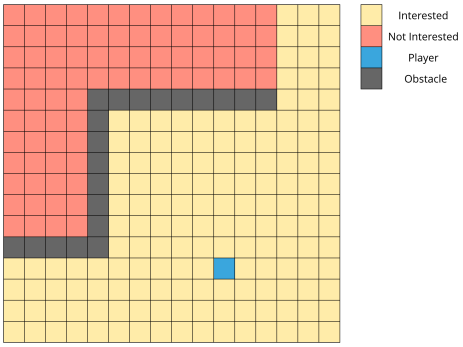

Figure 2.9: Tile interest map in a 3D game world

A *tile interest map* is built for every tile in the game world, which takes all obstacles into account. This can be a relatively expensive process and is typically pre-computed and stored in memory providing fast lookup times. An example of a tile interest map is shown in figure 2.9 in which the player (blue) is not interested in any of game objects on the red tiles as the view of those tiles is obstructed by the obstacle (grey).

2.3 Closely Related Projects

There proposed project can be broken up into three main areas - NDN, video game development and video game networking as seen in figure 2.10.

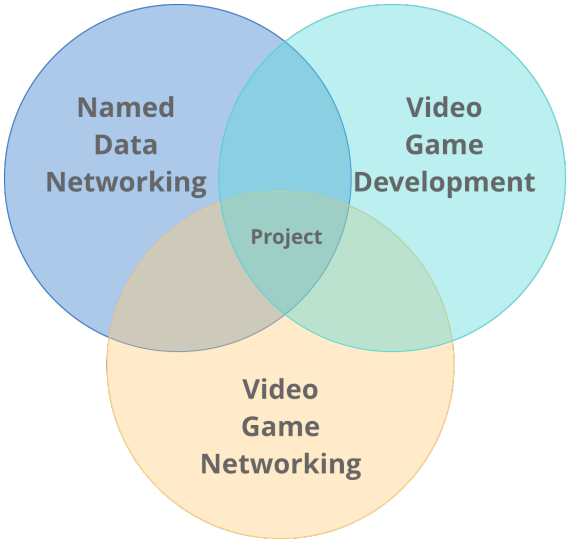

Figure 2.10: Core areas associated with the research project

As such, 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 [53]

Egal Car 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. Egal Car splits the data required for the game into three distinct categories. The first represents static, immutable, unchanging data which does not need to be synchronized, such as terrain and car assets. The second is data required for asset creation, such as when a new player joins the game. The third category represents state synchronization data, which can be tied to certain game entities or can be global for a particular instance of the game.

Egal Car made use of the now deprecated CCNx Sync protocol [34] for *dataset synchronization (DS)*. CCNx Sync provides **reliable and unordered** DS and was used for asset discovery as assets are entirely independent of one and other, meaning the ordering of asset discovery is not important.

However, as Egal Car's state updates are snapshots in time, CCNx sync could not be used as the ordering of state updates is critical to the consistency of the game. Egal Car made use of NDN's standard Interest and Data primitives for state synchronization, along with a timestamp floor, allowing players to only accept updates which were newer

than what they had previously seen.

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 publicly source code available.

Matryoshka [54]

Matryoshka 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.

Matryoshka uses a two step synchronization process within each partition - the discovery step and the update step.

Every game player maintains a hash of the set of names representing the game objects it knows about in the player's current partition. Game players express Interests for the partition they are currently in, along with the digest of the set of game objects in that partition that they know about, to the partition's discovery namespace. Other players in this partition receive these Interests and if the received digest does not match their own digest, they respond with a Data packet containing the set of names they know about. This allows players to discovery game objects in their partitions. The name schema for the discovery namespace used in Matryoshka is shown in figure 2.11. Finally, players can periodically express Interests for the game objects in their partition, using the set of names they have discovered.

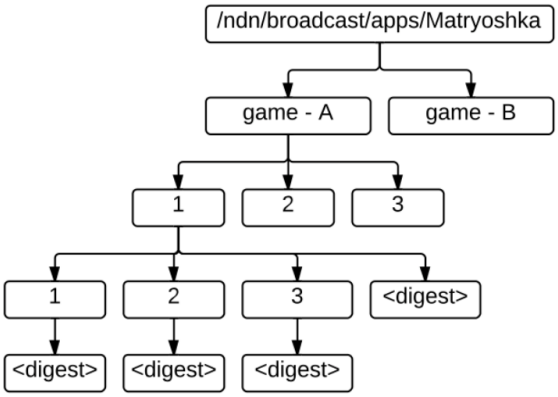

Figure 2.11: Matryoshka broadcast discovery namespace [54]

Matryoshka provides an interesting solution to the problem of interest management by having a deterministic method for constructing Interest names based on the player's game world location. The solution appears to be quite scalable by increasing the depth of the recursive partitioning to support smaller and smaller areas of interest. However, areas of interest cannot be infinitely decreased, which limits the overall scalability of the solution.

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 [55]

NDNGame 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 suggestion of using a hybrid architecture in which traditional host based communication is performed using IP, while content dissemination is performed using NDN is an interesting concept. However, the assumption that static game file content dissemination is anywhere near as challenging as the real time networking requirements of the MOG is flawed. Although, using a P2P like file sharing solution is ideal for large scale content dissemination, serving the static game files is an entirely orthogonal problem to building a highly scalable, low latency MOG experience. When a new game is released, there is likely to be a large demand for the static game files, while customers download the game. Although this is indeed an ideal use case for NDN, this spike can also be handled by

temporarily scaling the servers responsible for serving the static game files. 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.

3 Problem Statement

After conducting a detailed literature review of the fields of NDN and MOG networking, it became clear that the features offered by NDN could likely be exploited to build a highly scalable MOG. One of the main limitations in each of the closely related projects was the lack of an actual implementation and the lack of extensive testing. For this reason, a practical approach was desired to ensure that the actual performance of NDN in a MOG context could be empirically tested.

The research also showed that none of the existing libraries and frameworks offered by the NDN platform were a perfect fit for MOGs. As such, it became clear that a bespoke protocol would be required to build a scalable MOG using NDN.

The main goals for the design and implementation phases of the research are:

1. Design and implement a game which generates sufficiently diverse data such that each of the categories identified by the MOG data taxonomy are covered.
2. Design and build a P2P NDN based backend for the game which provides a consistent view of the game world to all connected players.
3. Allow players to interact with game world objects and other players.
4. Create a synchronization protocol which can detect and obtain remote game object updates in one round trip.
5. Create a comprehensive testing framework which can allow the game to be evaluated in a variety of scenarios. This should support testing using different topologies, with different numbers of players, with different game mechanics enabled and with different network optimizations such as interest management (IM) enabled and disabled.
6. The testing should be performed using an actual implementation running on actual hardware and not via a simulation (e.g. ndnSIM) or emulation (Mini-NDN) frameworks in order to provide as close to a real world scenario as possible.
7. The effects of enabling NDN features such as caching and network optimizations

such as IM should be studied in isolation to understand the exact effect they have on the performance.

8. The tests should be repeated using a variety of topologies which are sufficiently different from one and other.

9. Tests should be performed using a large number of players to investigate how the game scales.

4 Design

With the background research section of the project completed, the main focus shifted to designing the software required to facilitate the evaluation of NDN in a MOG context. This section outlines the design for the game, the backend module and several of the network optimizations which will later be implemented.

4.1 NDNShooter - a 2D, Top Down, Shooting Game

Although a variety of open source games of various complexities exist, building a new game from scratch was chosen over adding a networking module to an existing game for the following reasons:

1. The planned scope of the frontend of the game was very small, meaning the time investment to build the frontend of the game would not be substantial, in comparison to the rest of the project.
2. Reading and understanding a large code base is often more difficult than writing the code from scratch. Certain aspects of an inherited code base could also be misunderstood or overlooked, which has the potential to cause major problems in a research context.
3. Designing and implementing the game from scratch would allow for a deeper understanding of the overall system.
4. Although the networking aspects of the actual game are decoupled from the frontend game design, it is possible that building the game from scratch could lead to interesting questions arising when considered from an ICN perspective. For example, there may be optimizations that can be made to the frontend when targeting an ICN based backend and these optimizations would never be explored if an existing game was used.
5. Depending on the available time, the game could grow in complexity to support other features which are interesting from a networking perspective.

4.1.1 Design Requirements of NDNShooter

The style of game chosen was a simple 2D, top-down game in which a player could move an avatar around the game world. As the design of the actual game was not of interest to the research, NDNShooter was kept as simple as possible. The key purpose of the game was to provide a source of real world MOG traffic, enabling the study of NDN in this context. To this end, a list of requirements for NDNShooter was decided upon and contained the following:

1. The player must be able to move their local avatar around the game world.
2. The player must be able to see remote players moving around the game world in realtime.
3. The player must be able to perform actions which cause the game world to change for all players.
4. The player must be able to interact with local and remote game objects and have the updates propagate to remote players.
5. The player must be able to interact with remote players and the interaction must be visible to all remote players.

NDNShooter was designed to meat each of the requirements defined above and a screenshot from the game is shown in figure 4.1.

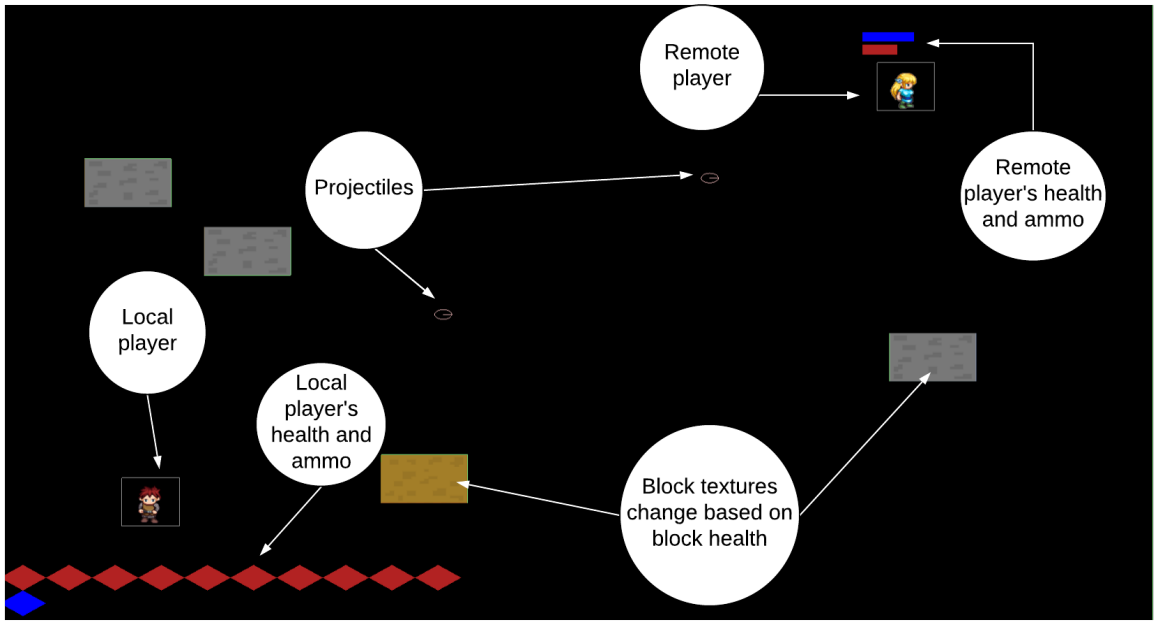

Figure 4.1: NDNShooter - a 2D, top down game developed to facilitate research into MOGs using NDN

NDNShooter contains both local and remote players, both of which can freely move around the game world, satisfying number 1 and 2 of the design requirements.

Players can also place blocks in the game world, which are seen as yellow and grey rectangles in figure 4.1. These blocks are visible to all players, satisfying design requirement number 3.

Blocks placed in the game world are given an initial amount of health and players may attack these blocks by walking up to them and pressing the left mouse button, or by shooting projectiles at them using the right mouse button. If a projectile hits a block, player or the game world boundary, the projectile is consumed and removed from the game. Provided the attack or projectile hits a block, the block's health will decrease by one. The texture used to render the block is dependent on the health of the block. This is seen in figure 4.1 as some of the blocks are grey in colour and some are yellow in colour. Upon successfully attacking a block with a single health point remaining, the block is also destroyed. This aspect of the game provides players with a means to interact with both local and remote game objects, satisfying requirement 4.

The red and blue diamonds seen in the bottom left corner of the screen in figure 4.1 indicate the local player's health and ammunition respectively. Players may attack other players using the attack mechanisms described previously. Upon shooting a projectile, the player's ammo is decreased by 1, and upon successfully attacking a player, the attacked player's health is decreased by 1, satisfying design requirement 5. The remote player's health and ammunition are also visible to local players. These are shown above the remote player's avatar as red and blue bars respectively. As seen in figure 4.1, the remote player's health and ammo are both partially empty, indicating the player has been hit by a number of attacks and has also shot projectiles.

4.2 NDNShooter Data Taxonomy

The taxonomy for MOG data is outlined in section 2.2.2. The proposed game design was examined to ensure each of the types of data outlined in the taxonomy were represented. The taxonomy of MOG data, along with the corresponding data in NDNShooter is shown in figure 4.2.

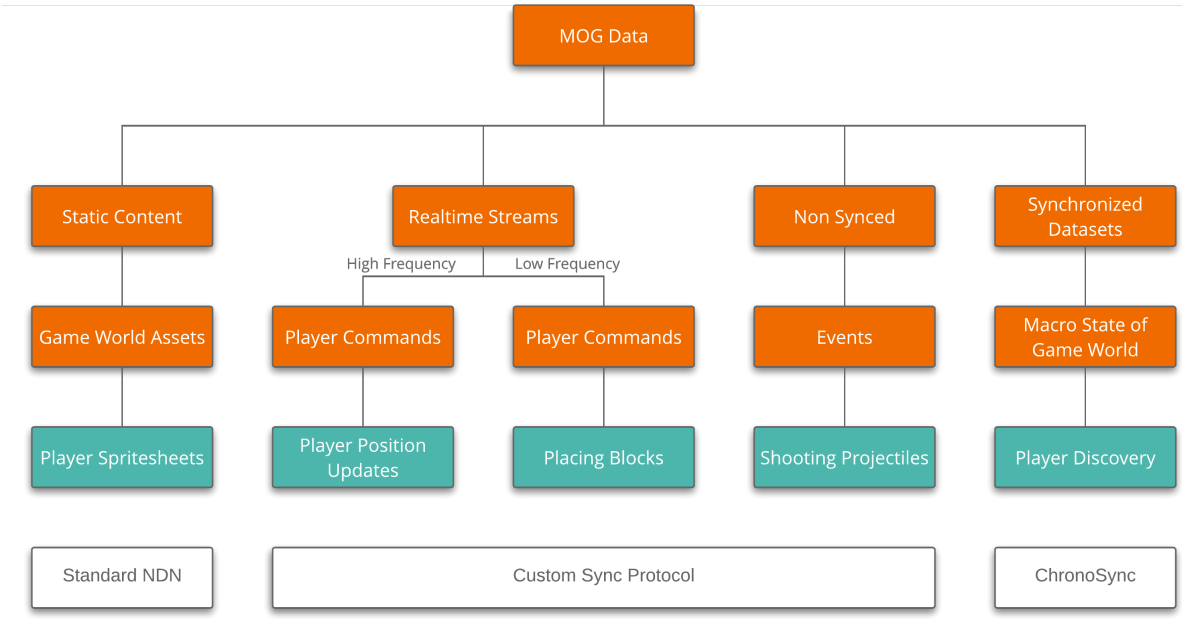

Figure 4.2: Taxonomy of data found in MOGs (orange), with the corresponding data in NDNShooter (green), and the protocols used to synchronize the data in NDNShooter (white).

As shown in figure 4.2, each of the data types produced by NDNShooter fit into one of the categories defined by the MOG data taxonomy, and an description of each is given below.

Static Content

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

As shown in figure 4.2, realtime streams are further subdivided into those which are high frequency and low frequency.

As players are free to roam around the game world, 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 As described earlier, one form of attacking is through shooting projectiles. 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 through the Interaction API (see section 4.5). 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 Player discovery is an 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 a extremely low frequency and can use a strict, slow protocol, to ensure players are discovered correctly.

Due to the variation in the data found in MOGs, multiple synchronization protocols are used, as shown in 4.2. These are discussed in the sections 4.3, 4.4 and 4.5.

4.3 Player Discovery

As shown in the taxonomy of NDNShooter's data (figure 4.2), the problem of player discovery is one of dataset synchronization (DS). As discussed in section 2.1.15, a variety of DS protocols exist as part of the NDN ecosystem. These protocols all require multiple round trips to fetch updated data, meaning they are not suitable for use with high frequency data such as that found in MOGs. However, the dataset associated with player discovery is updated very infrequently in NDNShooter. Similarly, for player discovery, the consistency of the dataset is far more important that than the latency associated with updating the dataset. As such, an existing solution for DS can be used for player discovery.

ChronoSync was chosen for player discovery as it is part of the NDN Common Client Libraries specification [22], meaning it is available in all of the supported languages. ChronoSync has also been around since 2013, meaning it is well documented and tested. Although ChronoSync contains some major limitations, as outlined in the discussion on PSync (see section 2.1.15), none of these limitations will cause any issues in the context of player discovery.

There are only two input parameters required for the naming schema used in NDNShooter - the *gameId* and the *playerName*. The *gameId* is chosen ahead of time and allows the player to choose the instance of NDNShooter they wish to play in. Thus, for player discovery, the only value that needs to be discovered to provide access to all data produced by a player is the *playerName*. This means the dataset synchronized by the player discovery mechanism is a set of strings, representing the the *playerNames* of all connected players.

As outlined in section 2.1.15, ChronoSync requires a broadcast namespace under which all nodes can produce *SyncInterests* and *SyncReplies*. These are used by participants to detect dataset changes and to inform others of the *name* of the data which has been added. The broadcast namespace used in NDNShooter is */<game_prefix>/discovery/broadcast*. As discussed in section 2.1.7, the forwarding strategy selected for a given namespace can be critical to the **correctness** of an application and is not only a network optimization choice. As all nodes must be informed of all updates to the dataset, the forwarding strategy for this name space must be *multicast*, which provides the broadcast functionality.

The final component of player discovery is the name used for fetching the updated player discovery data. Recall that ChronoSync nodes satisfy the *SyncInterest* with a *SyncReply* Data packet which contains the **name** of the Data packet to fetch to retrieve the update. In NDNShooter, the player discovery data is named */<game_prefix>/discovery/<player_name>*. The node who is responsible for publishing under this namespace will respond with the set of *playerNames* it currently knows about.

An important note here is the apparent redundancy in subsequently fetching the *playerName* using the discovered *playerName*. The reason player discovery was designed in this way, was to support future additions to the player discovery packet, without requiring changes to the implementation. For example, the player discovery data packet could be easily extended to include the team to which the discovered player belongs.

4.3.1 Benefits

The main benefit of using ChronoSync for player discovery was convenience. ChronoSync provides an easy to use API which is available in all of the NDN Common Client Library implementations, and due to the characteristics of the player discovery data, NDNShooter is not hindered by the limitations inherent in the ChronoSync protocol.

However, the current player discovery mechanism is naive in that it is performed globally across all players in a given game instance. Matryoshka (see section 2.3) uses an elegant solution for player discovery by only discovering players in a specific region of interest. However, in comparison to the other data types outlined in the MOG taxonomy (see section 2.2.2), player discovery is an extremely light weight task. Currently, the only data required by NDNShooter for player discovery is the player's name. Thus, even in the case of hundreds of game players, the size of the player discovery Data packets remains small. Similarly, the frequency at which the player discovery dataset changes is extremely low, relative to other categories of data in the taxonomy. This enables the use of a stricter, slower protocol such as ChronoSync.

The intended maximum number of players in a given instance of NDNShooter would be on the order of hundreds. This allows the player discovery protocol to be performed globally. However, if the game was to support thousands of players in a given instance, it is likely that a more complex protocol such as that employed by Matryoshka would likely be required.

4.4 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.

4.4.1 Motivation

Although several protocols exist for synchronizing datasets over NDN, there are some fundamental differences between the requirements for a distributed DS mechanism and game object synchronization in MOGs. The main difference is the priority of **low latency** over stricter consistency and ordering.

A common feature of the existing DS protocols is that they act as notification systems, informing participants of updates to the dataset and how to fetch those updates. It is up to the participant themselves to actually fetch the updated data. This approach

does provide benefits in the context of DS in that the scope of the protocol is reduced to notifying participants of updates, and participants then have the *option* to fetch the data, meaning they can ignore uninteresting updates. However, these benefits come at the cost of having two perform a second Interest / Data exchange to **obtain** the updated data. This has the effect of approximately doubling the round trip time of receiving updates, which is a major issue in the context of MOGs were latency is critical. Thus, a primary design goal of a game object synchronization protocol would be to achieve synchronization in a single Interest / Data exchange.

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 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.

4.4.2 Name Schema

One of the most important aspects of designing an application or protocol which uses NDN is the *name schema*. As discussed previously, 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 4.3.

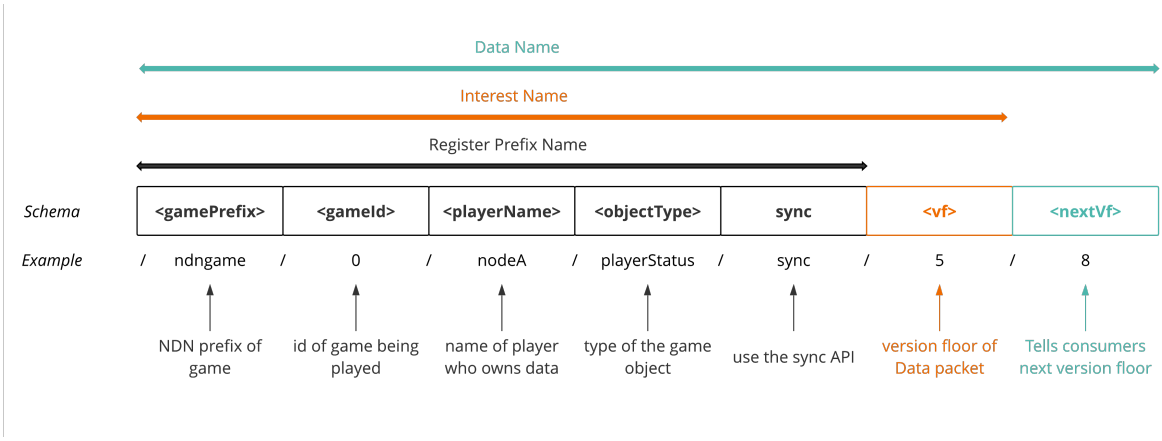

Figure 4.3: Name schema of NDNShooter's game object sync protocol

As seen in figure 4.3, 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.

gamePrefix

This component is used to target NDNShooter in the global NDN namespace.

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.

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 (see section 4.3).

objectType

This specifies the type of the game object in question. In the current implementation of NDNShooter, there are currently three possible values for this component:

1. *playerStatus* refers to the status of a player which includes information such as the player's position in the game world, velocity vector, health and ammo.
2. *blocks* refers to the set of active blocks in the game world that were placed by the player.
3. *projectiles* refers to the projectiles which the player has previously shot.

sync

This specifies that this packet is for use with the sync API as opposed to the interaction API (see section 4.5).

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.

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 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.

4.4.3 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 it's NFD. This is done using the *registerPrefix* call provided by the NDN CCL.

$$nodeA\ registers\ prefix:\ /ndngame/0/nodeA/playerStatus/sync$$

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 object's 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.

$$nodeB\ expresses\ Interest:\ /ndngame/0/nodeA/playerStatus/sync/0$$

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 it 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:

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

4.4.4 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 [53].

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 outlined in section 2.1.12, 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 ex-

pressed 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.

4.5 NDNShooter Interaction API

As discussed in section 2.3, one of the limitations of EgalCar was that game objects could not interact with one and other. In order to provide a basic means for interacting with remote game objects in NDNShooter, a simple API was developed. Any game object that is to support interaction must be uniquely addressable and is thus given an *id* which need only be unique across game objects of a certain type, belonging to a certain player.

4.5.1 Name Schema

As shown in figure 4.4, the interaction API uses a similar name schema to the sync protocol. The first four components are identical and are used to target a certain type of game object, belonging to a certain player, in a certain game instance. The fifth component is the keyword *"interact"* and this indicates that this Interest is for use with the interaction API. The sixth component is the *id* of the game object that the player wishes to interact with. Finally, the seventh component is the *command*, a specific keyword which defines the type of interaction to perform. The possible values for the *command* component differ depending on the *objectType* being targeted. For example, blocks can support being attacked or healed using the *attack* and *heal* commands respectively.

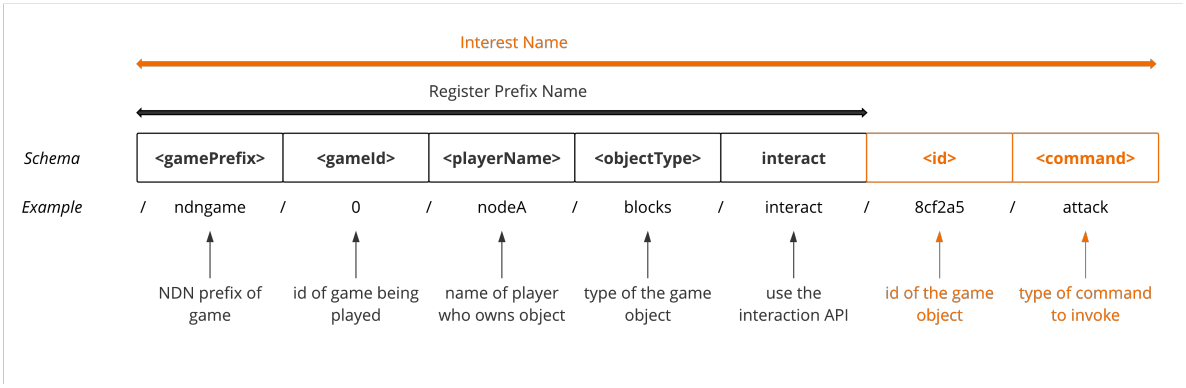

Figure 4.4: Name schema of NDNShooter's game object interaction API

The interaction API is used by players wishing to interact with game objects for which they do **not** hold the primary copy. Depending on the semantics of the game, the holders of the primary copies can choose to blindly accept these interactions, or can validate them using game specific logic. In NDNShooter, these updates are blindly accepted, as the game employs a *favour the shooter* approach, meaning events that occur which appeared consistent from the point of view of the player causing the event (e.g. the person shooting) are accepted as valid. This approach is somewhat naive and very susceptible to cheating, however this is beyond the scope of the current research.

Upon receipt of an interaction Interest, the holder of the primary copy, which is simply the owner of the game object in NDNShooter, can update the game object. This causes an update to be published through the sync protocol, allowing all interested players to see the result of the interaction.

4.5.2 Limitations

One of the challenges associated with NDN is that Interests cannot be easily parametrized without causing the Interest name to become more and more complex. For example, if the damage associated with an attack was variable, this would require yet another component added to the name to represent the damage value. This approach is feasible for a small number of parameters, but can get quickly out of hand in the case of parameters which are optional or contain nested fields.

Similarly, this type of scenario doesn't fit neatly into the abstraction provided by NDN, in that there is no Data associated with this Interest. Players simply express these interaction Interests and no Data packet is ever expected. The effects of this can be minimized by setting the Interest's timeout value to 0. This means it will not waste valuable resources by sitting in the PIT of intermediate routers once it has been forwarded.

An alternative design for the interaction API is to have primary copy holders maintain outstanding Interests for Data packets which represent **interactions** with the **game objects they own**. For example, nodeB could interact with a game object owned by nodeA, by satisfying nodeA's outstanding interaction Interest with a Data packet which contains the *id* of the game object nodeB wishes to interact with, along with any arbitrary data parametrizing the Interest.

However, the obvious drawback with this approach is that every player would need to register a prefix for the interaction Interests of every other player in the game. Thus, it appears a solution like this has limited scalability, though further investigation would likely be required which is beyond the scope of this research.

4.6 Dead Reckoning

4.6.1 Consumer Side PositionExtrapolation

As outlined in section 2.2.4, *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.

Another key component of DR is the convergence algorithm used. Complex DR convergence algorithms are an optimization to the *experience* associated with playing a MOG, rather than the networking performance of the MOG. As this research focuses on the networking performance of MOGs using NDN, the simplest form of a convergence algorithm will be used, in which game objects simply snap to their updated game positions.

4.6.2 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 (see section 2.1.2), and the name schema of the sync protocol makes no reference to consumer names (see section 4.4.2), 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 give time.

4.7 Interest Management

As discussed in section 2.2.5, *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.

As 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 4.5.

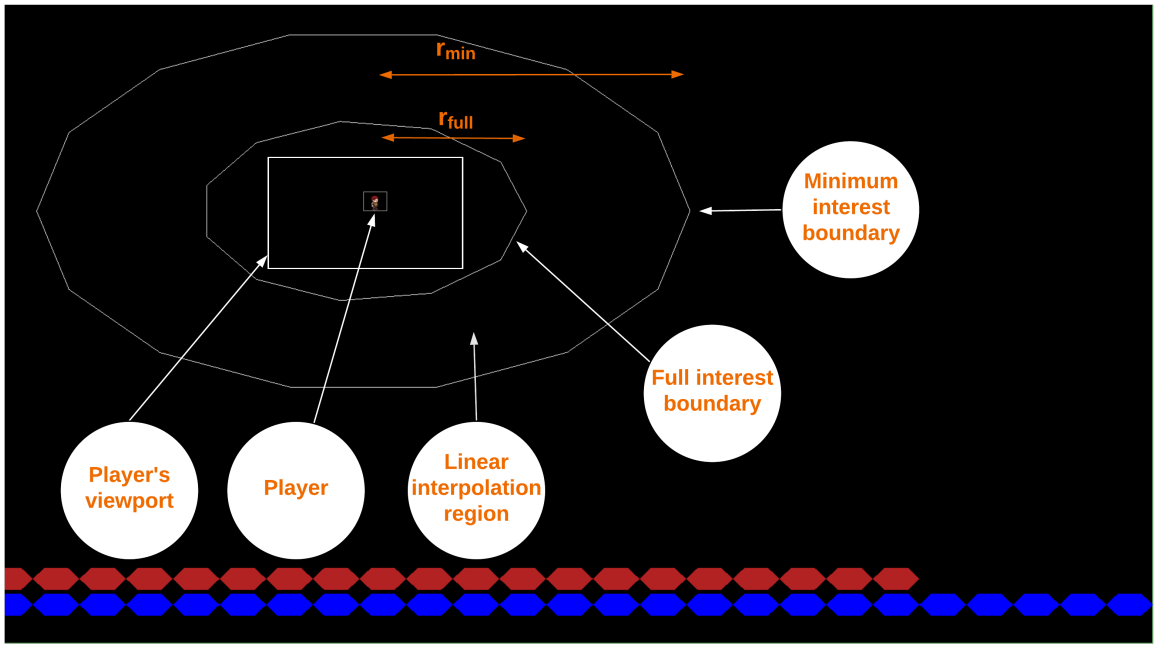

Figure 4.5: Parameters associated with NDNShooter's IM system

NDNShooter's IM system requires three parameters:

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.

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.

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_SLEEPseconds.

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:

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

4.7.1 Discussion

The inclusion of a 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.

As outlined in section 2.2.5, there a several more complex IM systems which can be used. However, most of these are targetted at 3D games which enable the IM system to take the actual game world into account. In NDNShooter, the player is presented with a 2D, top down view of the game world, meaning that game objects cannot be hidden behind other game objects. Thus, in this case, a simple distance based IM system suffices and will be a major boon to the scalability of NDNShooter.

5 Frontend Implementation

This section outlines how the frontend of NDNShooter was built, the libraries and frameworks used, and a description of several interesting implementation details that arose.

5.1 LibGDX

LibGDX [42] was used as the primary game development tool. LibGDX has an easy to use API and provides a variety of useful features such as drawing basic shapes to the screen, rendering textures, animation, orthographic cameras, input controllers and asset management. LibGDX produces an executable for each target platform, which in the case of NDNShooter was only a desktop environment such as Windows or Linux. LibGDX also provides access to the Box2D [56] physics engine which handles all of the physics in NDNShooter.

LibGDX makes heavy use of the Java Native Interface. This provides access to high performance C++ implementations of many performance critical methods such as those used for rendering.

5.2 Ashley - Entity Management System

Although LibGDX provides the basic functionality for rendering and Box2D provides all of the physics functionality required, an *entity management system (EMS)* can aid in writing efficient and reusable code in a video game setting. The EMS used in NDNShooter is Ashley [57].

5.2.1 Ashley's Architecture

Ashley uses four abstractions to simplify the management of entities in NDNShooter - *components*, *entities*, *systems* and *engines*.

Components

These are basic data structures which contain no logic whatsoever. Components contain the data required for an entity to implement some tangible game feature. For example, the *RemotePlayerComponent* contains all of the data required for an entity to be a remote player such as the NDN name which is used to fetch remote updates and the current version floor of the remote player. Another example would be the *AnimationComponent* which contains all of the data required to perform animation for this component, such as the list of frames to use for the animation.

Entities

Entities are containers for components. Each entity can have multiple components, but may only one instance of each type of component. For example, when the player discovery mechanism finds a new player, an entity is created which represents the remote player. However, entities alone are entirely generic from the point of view of the engine and the fact that this entity represents a remote player is entirely semantic. Thus, what makes this entity a remote player is the components which it contains. Remote players require several components, including a *RemotePlayerComponent*, an *AnimationComponent*, a *CollisionComponent*, and many others. As with components, entities contain no application logic and are simply wrappers for a set of components which make logical sense from the game's perspective (e.g. remote players).

Systems

Systems are at the core of Ashley and they perform application logic on entities which contain components that match a certain criteria. These component criteria are known as *families* and the first step in writing a system is specifying what family this system should operate on. For example, the *RemotePlayerUpdateSystem* is the system which interacts with NDNShooter's backend module to check for updates of remote players. Obviously this system should only operate on entities which represent remote players. However, from Ashley's point of view, every entity is just a generic entity which contains a set of components. Thus, the *RemotePlayerUpdateSystem* operates on the family of entities which contain *RemotePlayerComponents*.

Similarly, the *PhysicsSystem* only operates on entities which have a *BodyComponent*. The *PhysicsSystem* is responsible for stepping the physics world used by Box2D and *BodyComponents* contain the Box2D specific data associated with each entity.

Systems provide a clean way of splitting out the logic required by game objects into independent pieces which can be easily reused. They also allow for easy creation of new types of game objects. For example, when adding projectiles to NDNShooter, a new

ProjectileComponent was created which contains projectile specific data **only**. To enable physics for projectiles, *BodyComponents* were also given to projectile entities. Similarly, to enable collision detection between projectiles and other collision aware entities (e.g. players and blocks), a *CollisionComponent* was also given to entities representing projectiles. In doing so, physics and collision logic was given to projectiles, without requiring any changes to the underlying systems responsible for physics or collision detection.

Engine

The engine is the orchestrator behind Ashley and manages all of the entities in the game. When an engine update is invoked, the engine runs through each of the defined systems and passes them the set of entities which meet the family criteria they specify. The systems are executed in the order in which they are registered with the engine, which is a critical aspect of Ashley that must be carefully considered. Ashley also offers entity listeners, which are invoked when entities of a specifiable family are created or destroyed.

Ashley contains several engine types and NDNShooter uses the *PooledEngine*. This engine allows entities and components which have been previously removed to be reused, offering better performance than repeatedly garbage collecting and recreating objects as they are removed and added to the engine respectively.

5.2.2 Entity Representation in Ashley

A conceptual view of a remote player entity, the associated components, and the systems which operate on the remote player entity is shown in figure 5.1.

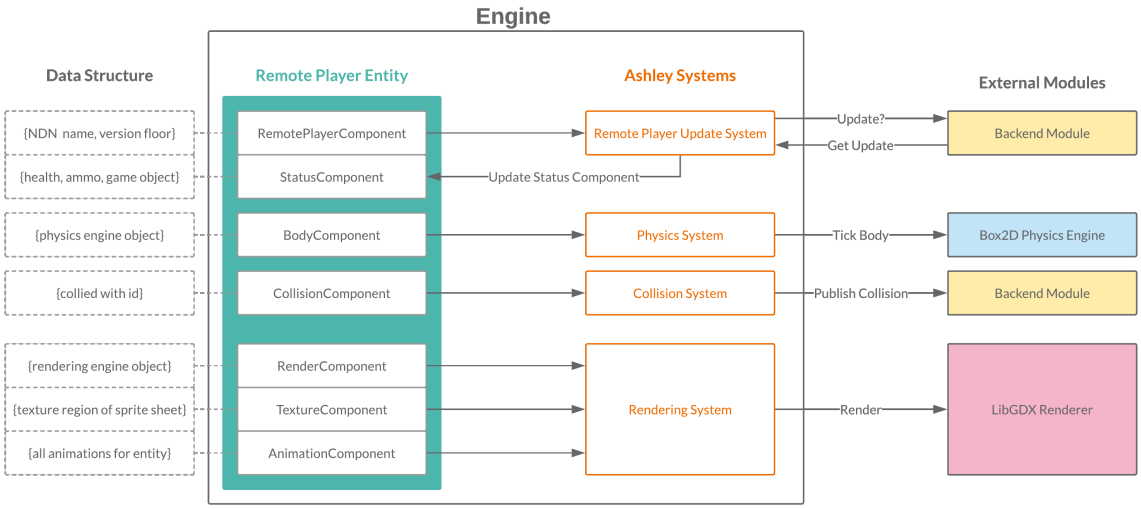

Figure 5.1: A simplified version of a remote player entity in Ashley

As with most entities in NDNShooter, remote player entities have a *BodyComponent* enabling Box2D physics , a *CollisionComponent* enabling collision detection and a *RenderComponent*, *TextureComponent* and *AnimationComponent* which are used to display the entity.

Remote player entities also have a *RemotePlayerComponent* and a *StatusComponent*. As previously discussed, the *RemotePlayerComponent* contains the data required to fetch updates for the remote player. The *StatusComponent* contains data specific to players such as their health, ammo and game object (position and velocity).

On each engine update, the Ashley engine invokes the *RemotePlayerUpdateSystem*. The engine passes each of the remote player entities to the system, using the family specified in the *RemotePlayerUpdateSystem* which requires entities which have a *RemotePlayerComponent*. The system then communicates with NDNShooter's *backend module* which implements the game object sync protocol as described in section 4.4. The *RemotePlayerUpdateSystem* checks for updates with the *backend module* using the NDN name and version floor contained in the *RemotePlayerComponent*. Finally, if the there is an update for the remote player entitiy, the system reconciles the local copy of the player using the corresponding reconciler (see section 5.3).

A similar strategy is used for local players, with the main difference being that the system informs the backend module of updates to the local player and these updates are then disseminated to consumers. Local players also require extra systems and components to enable the player to control the player using the keyboard and mouse.

5.3 Game Object Converters

A common requirement for the implementation is converting Ashley representations of game objects into bytes that can be sent over the wire. This is required when the game engine detects that an update should be published for a certain game object. Similarly, the bytes received over the wire must be converted back into Ashley representations of remote game objects. As such, the logic for these conversions was abstracted out of the Ashley systems and was built and tested independently.

As discussed in section 5.2, many of the entities share logic (systems) and data (components) for common functionality such as physics and rendering. Thus, converters were written in a reusable manner to enable code sharing. The most basic converter is the *GameObjectConverter* which is in turn used by the *BlockConverter*, *PlayerConverter* and *ProjectileConverter* which implement conversions for blocks, players and projectiles respectively.

Each of the converters described above expose two public methods - *reconcileRemote* and

protoFromEntity.

reconcileRemote

This method implements all of the steps which must be taken on receipt of a remote update to the entity in question. The converters contain a reference to the Ashley engine, meaning they can obtain and update the entity associated with the remote update.

For example, the *reconcileRemote* method of the *PlayerConverter* performs the following:

1. Reconciles the *BodyComponent* and *RenderComponent* of the remote player using the *GameObjectConverter*. This is the point in NDNShooter where an arbitrarily complex dead reckoning convergence algorithm could be used (see section 2.2.4).
2. Updates the player's health and ammo by updating the *StatusComponent* of the player
3. Updates the *StateComponent* of the player. This defines whether the player is currently walking, attacking or shooting a projectile and is used by the *RenderingSystem* and *AnimationSystem* to display remote players in the appropriate state.

Decoupling this method from the corresponding system is beneficial as it results in a single point in the code which defines exactly what happens to a given entity on receipt of a remote update.

protoFromEntity

As discussed in section 6.3, Google's Protobuf [58] was used for serialization of game objects. Protobuf requires *messages* to be defined ahead of time to enable serialization. As such, the *protoFromEntity* method is required for taking the entity representation of a game object and converting it into a Protobuf *message* which can then be serialized into bytes by Protobuf.

This method is also an ideal place to prune the entity representation of game objects back to the smallest amount of data that is required to be contained in the update. For example, player entities contain various timers which represent how long they have been in a particular state for. These are used to limit the rate at which players can perform actions such as shooting projectiles. However, as projectiles can only be shot by the primary copy holder, there is no need for this timer on the remote side meaning it does not need to be included in the updates sent to other players. Similarly, sending a player's sprite sheet in every update packet is obviously redundant and would result in bloated update packets, so this is also pruned at this point.

The *protoFromEntity* method is used by *update systems*. These include the *LocalPlayerUpdateSystem*, the *BlockUpdateSystem* and the *ProjectileUpdateSystem*. These are all Ashley systems which inspect the status of the entities they manage and determine whether or not an update to the entity should be published. If an update is required, the update system uses the *protoFromEntity* method associated with the entity to obtain the minimized Protobuf *message* representation of the entity. This is then given to *backend module* which publishes the update to the awaiting consumers.

5.4 Entity Creation and Deletion

As discussed in section 5.2, from the point of view of the Ashley engine, all entities are entirely generic. The understanding that a particular entity represents a remote player for example is entirely semantic and no compile-time checking is performed. As such, the semantic "type" of an entity is inferred based only on the components the entity has. Thus, the creation of these entities must be very strict and well defined to ensure entities contain the components they are supposed to.

Another issue with entity creation is the asynchronous nature of player discovery. As players require a *BodyComponent* which wraps a Box2D *Body* object, players may not be created **during an update** of the Box2D physics world. Otherwise, the native code backing the Box2D physics world will immediately throw an exception. The same issue occurs when entities with a BodyComponent need to be removed from the engine.

To solve these problems, entity creation and deletion were abstracted away to a separate *EntityManager* class. If an entity is to be created or deleted, clients can issue a request to this class. A new Ashley system was then written which informs the *EntityManager* when it is safe to perform all of the required entity creation and deletion.

The use of the *EntityManager* allows the exact specification for entities to be defined at a single point in the code base. This ensures all entities of a certain semantic type (e.g. remote players) are guaranteed to contain all of the components they require and to be fully formed and consistent by the time they are accessed.

6 Backend Implementation

One of the main goals when designing and building the backend of NDNShooter was to ensure it was **entirely** decoupled from the frontend. This means that an equivalent backend module can be built using an IP based stack, allowing for a direct comparison between NDN and IP in an identical scenario, though this was beyond the scope of the current research (see section 9.1).

This section outlines how the backend module for NDNShooter was implemented and includes information on how NDN was configured, the frameworks and libraries used, and a description of several interesting implementation details that arose during the development.

6.1 NDN Configuration

The design and implementation of NDNShooter requires careful consideration of the possible NDN specific configuration values that can be used. The configuration used for the NDN infrastructure impacts the performance, correctness and player experience of NDNShooter. There are three main areas to consider in this regard - *forwarding strategies*, *Interest packet parameters* and *Data packet parameters*.

Forwarding Strategies

As discussed in section 2.1.7, a *forwarding strategy* defines what the NDN forwarding daemon (NFD) should do when there are multiple next hops to choose from when forwarding an Interest. The most common forwarding strategies used by the NFD are *best-route* which forwards the Interest over the best performing next hop, and *multicast* which forwards over all of the possible next hops.

As outlined in section 4.3, the player discovery mechanism uses ChronoSync (a dataset synchronization protocol). As any of the participating nodes may write to the shared dataset, any of the participating nodes may satisfy the outstanding *SyncInterest*. Thus, the *SyncInterest* expressed must be forwarded to all participants and not just over the link which is currently performing the best.

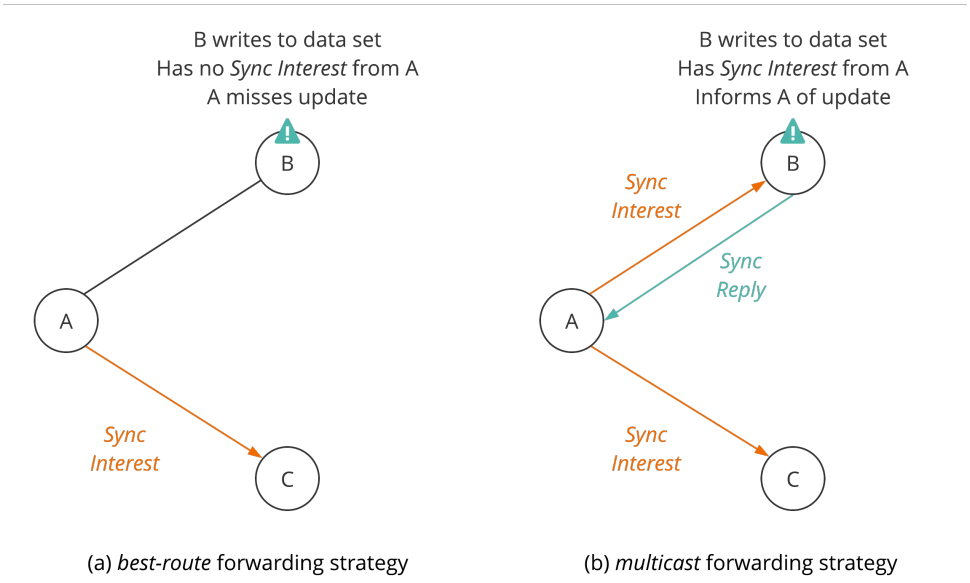

Figure 6.1: ChronoSync requires a broadcast namespace to ensure all participants receive update notifications

The necessity for a broadcast namespace in ChronoSync is shown in figure 6.1.

If the *best-route* forwarding strategy is used by node A's NFD, the *SyncInterest* generated by ChronoSync will only be forwarded over one upstream face. This is shown in figure 6.1 (a) in which node A's *SyncInterest* only reaches node C. If node B writes to the dataset, node B's ChronoSync module will satisfy the outstanding *SyncInterest* to inform participants of the update. However, node B did not receive a *SyncInterest* from node A, as it was only forwarded to node C, meaning node A will not be informed of the update.

However, if the *multicast* forwarding strategy is used, node A's NFD will forward the *SyncInterest* to both node B and node C as required, allowing node B to satisfy the Interest and node A to receive the update notification.

Thus, to ensure the player discovery mechanism works appropriately in NDNShooter, the forwarding strategy for the broadcast namespace was set to multicast on each of the players' NFD. This was done using the *NFD Control (nfdc)* [59] command line tool.

For all other namespaces used by NDNShooter, the forwarding strategy used was *best-route*. However, there are several versions of this forwarding strategy available in the NFD, the newest of which contains some extra functionality to handle consumer retransmission [60]. Unfortunately, this causes some problems will long term Interest aggregation which is a major benefit in the MOG context. For this reason, version 1 of *best-route* was used instead.

Interest Packet Parameters

As outlined in section 2.1.2, Interest packets contain a variety of parameters. Three of these are relevant to NDNShooter - *CanBePrefix*, *MustBeFresh* and *InterestLifetime*.

The *CanBePrefix* parameter specifies whether or not the name of the Interest packet can be a prefix of the name of the Data packet used to satisfy it. If set to false, the Interest can only be satisfied by Data packets with a name that **exactly matches** the name of the Interest. However, the sync protocol described in section 4.4 requires producers to append the *nextVersionFloor* to the Interest name. Thus, in NDNShooter, all Interest packets set the *CanBePrefix* parameter to true to enable this functionality.

The *MustBeFresh* parameter specifies whether or not the Interest can be satisfied by a CS entry that is no longer fresh. A Data entry in a CS is considered fresh until it has been in the CS for longer than the *FreshnessPeriod* specified in the Data packet. As MOGs require the most up to date version of a piece of data, the *MustBeFresh* parameter is set to true for all Interests produced by NDNShooter in order to not retrieve data from a cache which is stale.

Finally, the *InterestLifetime* defines the length of time the Interest will remain active before timing out. This is a tricky choice in NDNShooter as the sync protocol makes use of long lived outstanding Interests, meaning this value must be kept relatively high. For example, if a player doesn't move, there is no need for a producer to publish an update, meaning the outstanding Interests should be kept active for a relatively long period of time. However, if an Interest goes missing in the network, the consumers will wait until the *InterestLifetime* expires before re-expressing the Interest. Thus, a long value for *InterestLifetime* can cause gameplay problems in network conditions where packets are frequently lost. The current implementation of NDNShooter uses an *InterestLifetime* of 1 second in an attempt to minimize the number of re-expressed Interests which time out but were not lost, while simultaneously minimizing the impact of lost Interests.

Data Packet Parameters

As with Interest packets, Data packets also contain NDN specific parameters. The only parameter of interest in Data packets is the *FreshnessPeriod*. As discussed previously, this defines how long a Data packet is to be considered fresh. An important aspect of the *FreshnessPeriod* is that it is defined on a hop-by-hop basis. For example, if node A produces a Data packet with a *FreshnessPeriod* of 100ms, the Data packet in node A's CS will be considered fresh for 100ms. However, if this is forwarded to an intermediate router, the corresponding CS entry will be considered fresh for 100ms **after the router receives the Data packet**.

As MOGs require extremely recent data, the *FreshnessPeriod* used by NDNShooter is kept extremely low with a value of 20ms. However, the **actual** freshness of a piece of Data is dependent on the topology. Thus, a value of 20ms may cause players behind a long chain of intermediate routers to be receiving Data from caches that is actually quite old.

This approach sufficed for the topologies tested in section 8 which typically had a small number of intermediate routers. However, further research into this area would be required for wide scale deployment across varying topologies. Dynamic adjustment of the *FreshnessPeriod* based on the topology perceived by the producers may be beneficial in this regard and this is discussed further in section 9.1.

6.2 Version Floored Projectile Cache

As outlined in the taxonomy of MOG data (see section 2.2.2), one of the main categories of data in MOGs is *non synced* data. Typically, this represents short lived, *"publish and forget"* data such as events. In NDNShooter, the projectiles fall into this category. As projectiles typically last hundreds of milliseconds or a few seconds, there is no time or need to monitor and publish repeated updates to projectiles. Instead, they are given an initial state and this is disseminated to each consumer.

Unlike the other data found in NDNShooter, when consumers express an Interest for projectiles with a given version floor, they are interested in all of the projectiles which have been created by the publisher since that version floor. To enable producers to provide consumers with this data, a *version floored cache (VCF)* was developed.

The VCF is a fixed sized cache and allows insertion of items into the cache by value, replacing the oldest entry in the cache once full. Conceptually, the VCF can be thought of as a map which uses the version floor as the key and a list of events as the value. However, this implementation would require substantial repetition. Instead, the VCF was implemented as an array based, circular data structure, which only requires each of the projectiles to be stored a single time. By keeping track of the current minimum and maximum version floors contained in the cache, a *startIndex* and *endIndex* can be calculated for a given version floor. These indices are then used to extract the elements between the indices (while wrapping around if necessary), representing the new events since that version floor.

The possibilities on reading from the cache are as follows:

1. If a version floor lower than the lowest version floor contained in the cache is requested, the VCF simply returns all of the elements in the cache. This provides a catchup mechanism for new players who may request elements using a very old

version floor. The size of the cache in NDNShooter is tweaked such that valid projectiles will never be pushed out of the cache due to a lack of space. The size of the cache is closely related to the speed of the projectiles produced in NDNShooter, and the maximum allowable rate of shooting projectiles.

2. If a version floor larger than the highest version floor contained in the cache is requested, the VCF returns an empty set. This is understood by producers and they will defer satisfying the Interest until there is data available in the cache.

3. Otherwise, a subset of items in the cache is returned, representing the new entries since the version floor provided.

6.3 Protocol Buffers (Protobuf)

As described previously, a mechanism is required for serializing Ashley's entity representations of game objects, into representations which can be sent across the network. The mechanism used in NDNShooter is to translate the entity representation into Protobuf [58] *messages* which can be serialized by the Protobuf library in a very efficient manner. Thus, the design of these messages is an important aspect of the performance of NDNShooter's backend module.

Protobuf requires all *messages* to be defined in special *.proto* files. These are then compiled into the desired language (e.g. Java, C++) using the Protobuf compiler. The Protobuf framework supports a variety of features when defining *message* structures such as:

Typing	Fields are strongly typed and Protobuf provides standard types such as *int32*, *int64*, *float*, *string* etc.
Composition	*Messages* definitions can contain references to other *messages*.
Sequences	*Message* fields can be sequences of arbitrary length.
Maps	Fields can be maps which use Protobuf primitives or even other *messages* as keys and values.
Enums	Fields can be instances of types defined by enums.

The first *message* defined was the *GameObject* message. This contains the physical data associated with objects in the NDNShooter game world. Most of this data comes from the *BodyComponent* and *RenderComponent* of the Ashley entity. The producer's *GameObjectConverter* generates these messages and the consumer's *GameObjectConverter* reconciles the differences between the remote version and their local Ashley entity,

as described in section 5.2. The definition of the *GameObject* Protobuf *message* is show below.

```
message GameObject {
    float x = 1;
    float y = 2;
    float z = 3;
    float velX = 4;
    float velY = 5;
    float width = 6;
    float height = 7;
    float angle = 8;
    bool isFixedRotation = 9;
    float scaleX = 10;
    float scaleY = 11;
}
```

Code Listing 6.1: Protobuf Message representing NDNShooter game objects

However, *GameObjects* are not synced directly. Instead, *GameObject messages* are used as fields in more complex messages which are then synced. An example of a synced *message* is the *PlayerStatus message* which contains a *GameObject message* as discussed, and a *Status* message which contains information such as the entity's health and ammo. The definition of the *PlayerStatus message* is shown in code listing 6.2.

```
message PlayerStatus {
    GameObject gameObject = 1;
    Status status = 2;
}
```

Code Listing 6.2: Protobuf Message representing a player's PlayerStatus

Once all of the Protobuf *messages* are defined, they are compiled into actual Java code which provides access to a *Builder* to create an instance of the *message*, and a method to generate a byte array representation of the *message*. The byte array can then be used to create an instance of an NDN *Blob* [61], which represents the content of a Data packet.

6.4 Dependency Injection

As video games are a relatively complex piece of software, using *dependency injection (DI)* can drastically simplify the design of classes. DI refers to passing all of the dependencies of a class to the constructor of the class, instead of having the class instantiate the

dependencies itself. This facilitates unit testing of arbitrarily complex classes, as the dependencies of the class can be mocked and passed to the constructor of the class. Without DI, the instantiation of dependencies would be buried within the class logic meaning actual implementations of dependencies must be used.

Another major benefit to DI is that it mitigates the so called *"dependency hell"* problem. For example, if *Class A* requires an instance of *Class B*, *class A* must first obtain an instance of the dependencies of *Class B*, say *Class C* and *Class D*, in order to instantiate *Class B*. However, to instantiate *Class C*, *Class A* must first obtain instances of *Class C's* dependencies and so on. As outlined by Bloch [62], using composition is the favourable approach over relying on inheritance in large scale applications. Thus, *"dependency hell"* is a real problem in modern object oriented software.

Google's Guice [63] is a Java framework for using DI. Guice solves this problem by taking over the actual instantiation of classes on behalf of the programmer. Guice maintains a dependency graph which enables it to resolve dependencies required by classes. There must always be an entry class which requires no dependencies. This entry class can then be instantiated by Guice, and automatically injected into other classes which require it. This process can the repeat, allowing arbitrarily complex dependency graphs to be managed by Guice, allowing the programmer to simply inject all of the dependencies a class requires, provided the dependencies themselves used DI for their dependencies.

Due to the benefits outlined above, DI was used extensively and drastically simplified the management of dependencies in NDNShooter.

7 Testing Implementation

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.

7.1 Player Automation

In order to enable testing without actual people playing the game, a simple automation mechanism was developed. This was accomplished by creating a *InputController* interface. Two implementations of this interface were created - a *RealInputController* which actually reacts to user input via the keyboard and mouse, and an *AutomatedInputController* which simulates key strokes.

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 move-set. 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.

7.2 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. Docker allows for the creation of entirely self contained *containers*, which represent a *standardized unit of software* [64]. Docker containers run on a Docker engine, which is built for a specific host operating system such as Linux. However, the Docker containers themselves are entirely independent to the host operating system, meaning they run identically, regardless of the underlying platform.

A Docker *image* is used to define a Docker container. This contains **everything** that is needed to run the piece of software, such as libraries, frameworks and dependencies. A Docker image which contains everything required to run NDNShooter was developed based on Peter Gusev's [65] NDN Docker image. This enabled NDNShooter to be run on any machine which had Docker installed.

Docker Compose [66] is a tool which allows for the deployment of multiple Docker containers at once. This is done by defining Docker *Services* inside of a *compose file*. Each *service* definition in a Docker *compose file* contains the information for launching a number of containers such as what image to use and how many replicas of the image are required. For each of the topologies tested, a separate *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.

Docker also allows for the creation of virtual networks, over which Docker containers can communicate. The default network type is a *bridge* network, which enables containers to communicate with one and other, provided they are on the same host machine. This allowed the NFDs of each of the nodes (game players and intermediate routers) to communicate with one and other through the bridge network.

This was accomplished 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.

At this point, a single Docker Compose command could create all of the NDNShooter nodes required by the topology, as defined in the compose file, and run them on the host machine. 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 supports orchestrating clusters of separate Docker hosts using *swarms*. The services contained in a docker compose file can be deployed across a swarm using a Docker *stack*. The final requirement was to enable the Docker containers running on separate hosts to communicate with one and other. As discussed previously, *bridge* networks only allow containers running on the same **Docker host** to communicate.

Docker allows containers running on different Docker hosts to communicate by using an *overlay* network. The actual network is entirely managed by Docker and allowed all of the NDNShooter nodes to communicate regardless of what Docker host their container resided on.

When using swarms, Docker creates and uses an *ingress routing mesh* by default. This allows requests which reach any node in the swarm to be routed to an appropriate node who can service the request. For example, if a stack consists of a database container which exposes port 3306, and a web server container which exposes port 80, if a request for a web page (using port 80) is made to the database container, the ingress routing mesh automatically forwards this requests to the web server container. This is an extremely useful feature of Docker as it allows requests to be serviced regardless of the actual node the request is sent to. Similarly, this can also be used for load balancing requests across the swarm.

However, this behaviour is not desired in the case of testing NDNShooter, as Interests that are forwarded to a particular network alias, **must** arrive at the specific node they are forwarded to. Each of the nodes exposed the default NFD port of 6363, meaning Interests could actually arrive at different node to the one it was meant to be forwarded to. To overcome this problem, the endpoint mode of each Docker service required by the topology was set to *DNS-round-robin*, instead of the default *ingress routing mesh*.

An example topology containing 2 game players, and the corresponding Docker compose file used to test the topology are shown in appendix B.

7.3 NLSR Configuration

As defined in the previous section, each of the nodes used to test NDNShooter can 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.

The Docker overlay network serves as a means to create the communication links required by the NDN topology. This aligns with the vision of NDN being the *narrow waist of tomorrow's Internet*, with UDP/IP being one of the lower level protocols that NDN can in turn use for communication.

As described in section 2.1.10, NLSR 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 below:

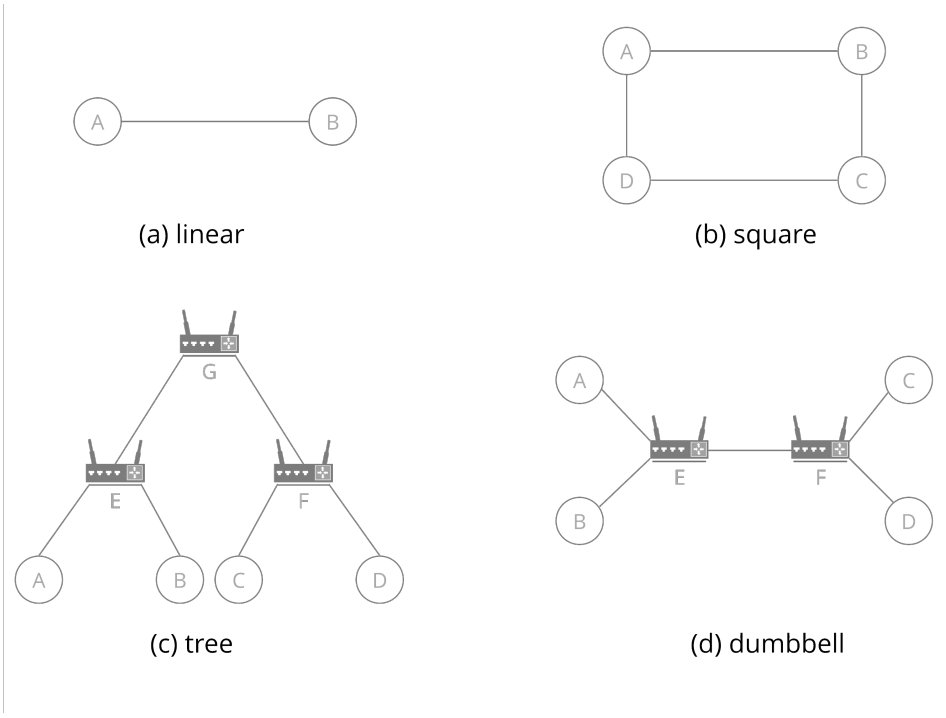

Figure 7.1: An example of the topologies used for testing NDNShooter

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.

7.4 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 [67] was used. This provides many useful classes for gathering data such as histograms, counters, timers and rate-trackers. All of the data tracked by this framework can be monitored in real-time using a web dashboard. However, to perform a more detailed analysis, the data was also periodically written to a CSV file.

Gathering these 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.

7.4.1 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 1,

$$RTT = t_{data} - t_{int} \tag{1}$$

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 2.

$$RTT_{eff} = (t_{data} - t_{int}) - t_{wait} \tag{2}$$

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 who's 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.

7.4.2 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 3, 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.

$$iaf_p = \frac{n_{seen\_p}}{\sum_{x \epsilon P} n_{exp\_x\_p}} \tag{3}$$

7.4.3 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.

7.4.4 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.

8 Evaluation

In order to validate the design and evaluate the performance of NDNShooter, several tests were performed in a variety of scenarios. As there a large number of parameters which impact the overall performance, each test was designed to isolate a single parameter to determine the impact the parameter has on the game.

As the performance of NDNShooter is heavily dependent on the NDN topology, all tests were repeated using several topologies where appropriate.

The performance is also impacted by the game mechanics which are enabled and the automation script used. As described in section 7.1, the automation script uses a fixed random seed to allow for a direct comparison between tests using the same topology.

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 identical to those gathered using the *PlayerStatus*, within a margin of error. Thus, as there is substantially more data available when considering the traffic generated by the *PlayerStatus* updates, this data type was favoured.

However, in all tests, **all game mechanics were enabled** and players were able to move, place blocks and shoot projectiles. Thus, all of the data used by NDNShooter was generated during these tests, but the *block* and *projectile* data is ommited 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 ommited from the figures as they offer no additional insight.

The evaluation is split into two stages - section 8.1 examines the impacts of tweaking parameters on the metrics discussed in section 7.4 and section 8.2 examines how NDNShooter scales in a real world scenario.

8.1 Metric Testing

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 defines the *freshness period* on a hop-by-hop basis.

Dead Reckoning (DR) Pub Throttling

This mechanism reduces the amount of updates a publisher will produce by maintaing a view of the game object as seen by the consumers. This is outlined further in section 4.6. There are two parameters associated with this mechanism - whether or not it is enabled, and the error threshold which results in a update being produced. The error threshold is a measure of the distance and uses *game world units (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. These parameters are discussed previously in section 4.7.

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 4.4, 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.

Metrics

The metrics used for the evaluation are the *round-trip-time (RTT)*, *Interest aggregation factor (IAF)*, *cache rate* and *position delta*, each of which are discussed in section 7.4.

Parameter-Metric Matrix

The *parameter-metric matix* is shown in table 8.1. This table shows the theoretical impact on each of the metrics when a parameter is **increased**. The *+*, *0* and - values in the cells represent an increase, no change and a decrease to the associated metric. Finally, the resulting impact on the overall performance of the game is indicated by the colour of the cell, where green, white and red indicate a positive, neutral and negative impact on the performance respectively.

Test Parameters	**Impact on Metric**			
	RTT	**IAF**	**Cache Rate**	**Position Delta**
Interest Timeout (ms)	0	0	0	+
Data Freshness Period (ms)	-	0	+	+
DR Pub Throttling Enabled	+	+	+	+
DR Pub Throttling Threshold (gwu)	-	+	0	+
IM Enabled	0	-	+	+
IM Full Radius, r_{full} (gwu)	0	+	-	-
IM Min Radius, r_{min} (gwu)	0	+	-	-
IM Max Sleep Time, MAX_SLEEP (s)	0	-	+	+
Pub Update Rate (Hz)	-	-	-	-
Pub Queue Check Rate (Hz)	-	-	-	-

Table 8.1: Configurable parameters and their associated impacts on the metrics. *+*, *0* and - represent an increase, no change and a decrease to the metric respectively when the parameter is **increased**. The colour of the cells represents whether this has a net positive (green), neutral (white) or negative (red) impact on the performance of NDNShooter

.

For all tests the following parameter values were unchanged:

$$Interest\ Timeout = 1000ms$$
$$Pub\ Update\ Rate = 30Hz$$
$$Pub\ Queue\ Check\ Rate = 60Hz$$

For convenience, the testing topologies discussed in section 7.3 are repeated in figure 8.1. A subset of these topologies were used for all tests.

Discuss why I chose these topologies

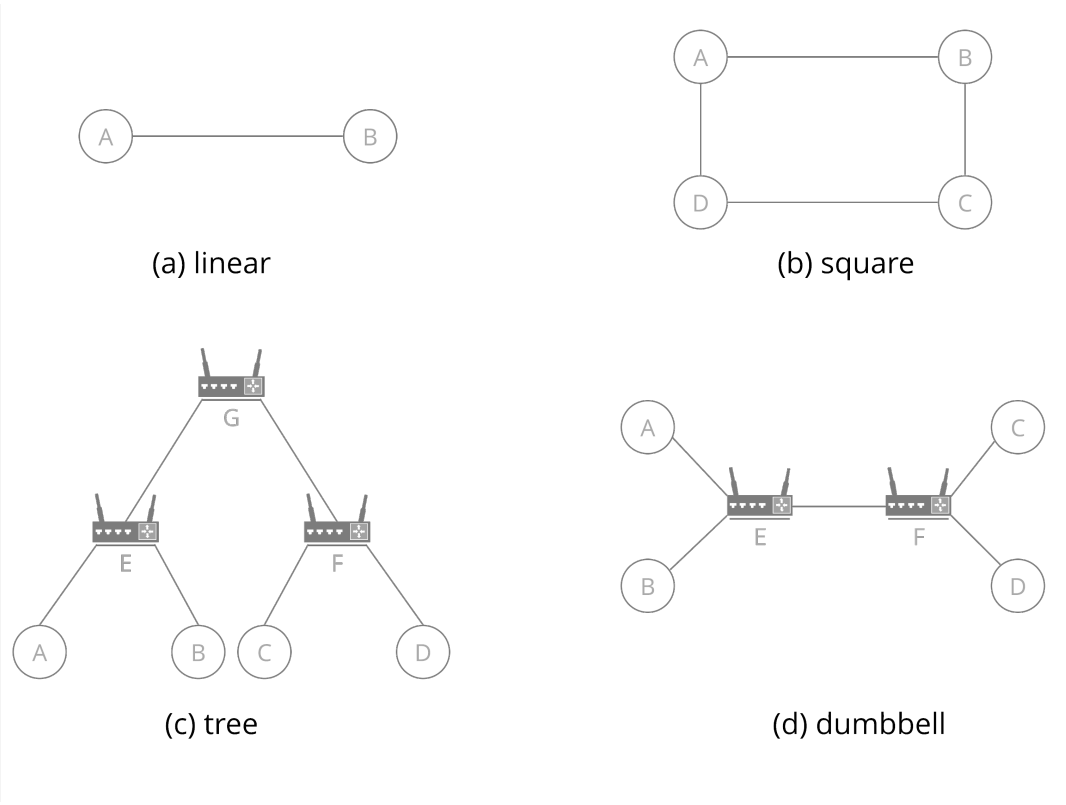

Figure 8.1: Topologies used in evaluation of NDNShooter

An important note to make here is in the magnitude of a *GWU*. GWUs are used to by LibGDX to remove the need to consider different resolution sizes. To provide some context, in NDNShooter, the width of a player avatar is 1 GWU as shown in figure 8.2

Figure 8.2: The width of a player avatar in NDNShooter is 1 GWU

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.

8.1.1 No Caching, no IM and no DR Publisher Throttling

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

Round-Trip-Times

The first metric examined was RTT. Each of the four topologies performed similarly in this regard. For this reason, the RTTs are only shown for the *dumbbell* topology, though the others can be found in appendix C. As seen in figure 8.3, 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.

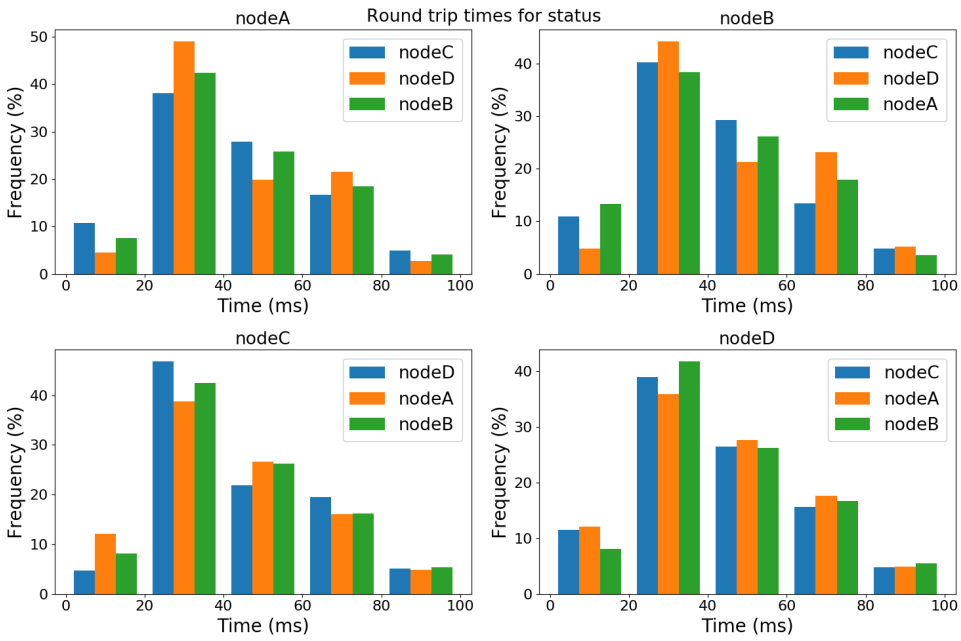

Figure 8.3: *RTTs* for *PlayerStatus* in the *dumbbell* topology

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 flooded, the *position deltas* should be extremely low in this case. Otherwise, the lack of network optimizations may effect performance sufficiently that the *position deltas* will be large.

As with the RTTs, each of the topologies performed very similarly in this regard. Again, only the histogram for the *dumbbell* topology is shown and all others are given in appendix C.

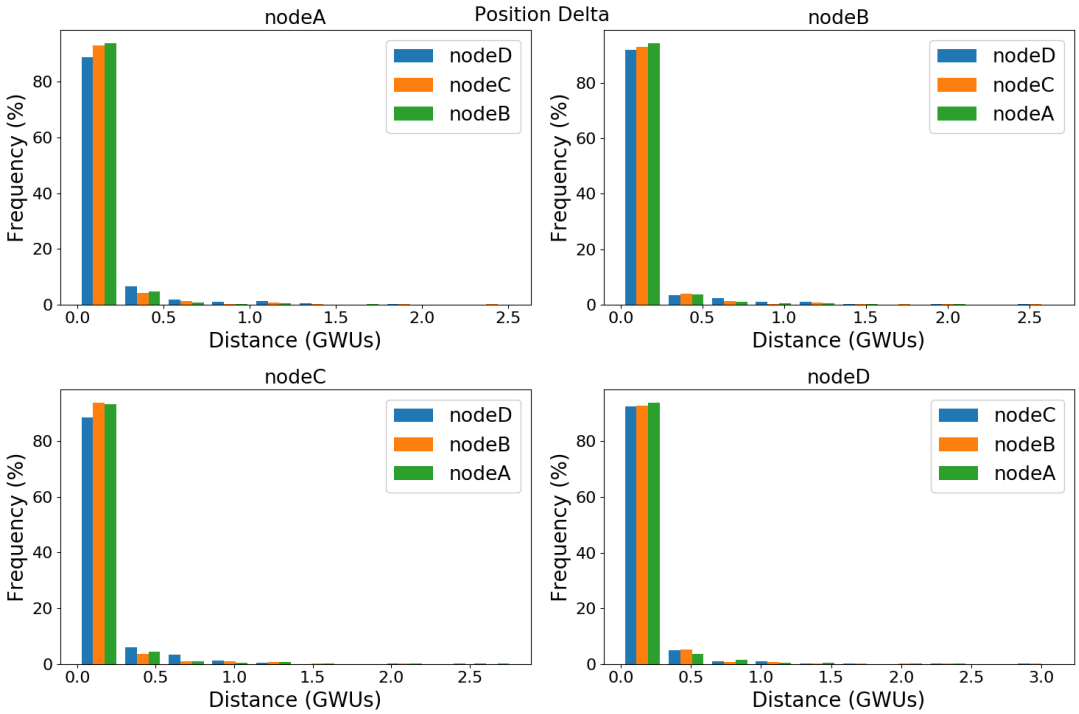

Figure 8.4: *Position deltas* for players in the *dumbbell* topology

The *position deltas* seen in figure 8.4 show that player positions are syncrhonzied very closely. The figure shows that almost 90% of the time, when a position update is received for a player, the dead reckoned position was within half a *GWU* of the new position. The figure also shows that the *position delta* can spike as high as 3 *GWUs*, although this is extremely rare. As these tests were performed on a single machine, this may be a result of CPU contention amonsgt the Docker containers.

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 8.5 and figure 8.6. 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.

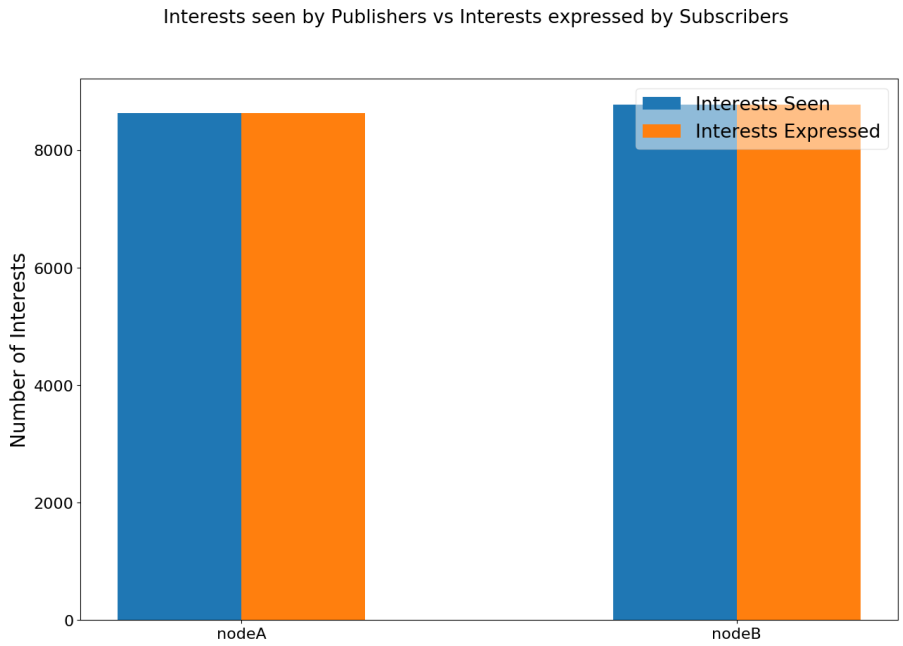

Figure 8.5: Interest aggregation for *PlayerStatus* in the *linear* topology. The *IAF* of both nodes is 1 as expected.

As expected, when there are only two nodes, no Interest aggregation occurs. Thus, the orange and blue bars match in figure 8.5 and both nodes had an *Interest Aggregation Factor (IAF)* of 1.

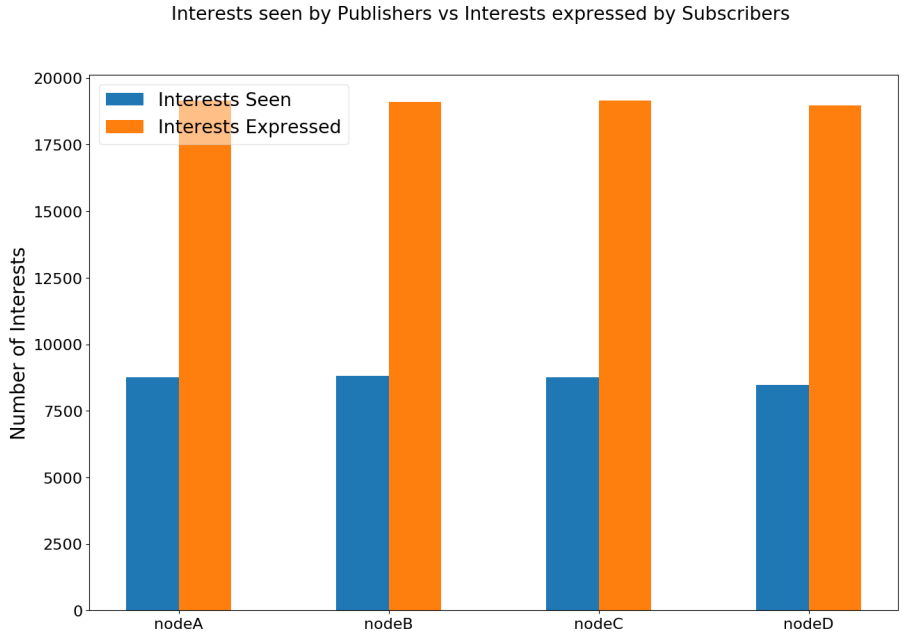

Figure 8.6: Interest aggregation for *PlayerStatus* in the *dumbbell* topology. The *IAF* of nodes A, B, C and D were calculated to be 0.45, 0.46, 0.45 and 0.44 respectively

The *dumbbell* topoology on the other hand shows that 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. Again, the other topologies performed very similarly and the graphs are provided in appendix C.

8.1.2 Effects of Enabling Caching

As outlined in section 2.1.12, caching can provide benefits to NDNShooter in the case where a node falls behind the Interest aggregation period, which is quite possible with an publisher update frequency of 30Hz. 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, with the only difference being that the Data packets were given an *freshness period* of 20ms. IM and DR publisher throttling both remained disabled for these tests and the *linear* topology was ommited as caching will not have any impact in the case of only 2 players.

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 8.7. The results for the *square* and *dumbbell* were very similar and can be found in appendix D.

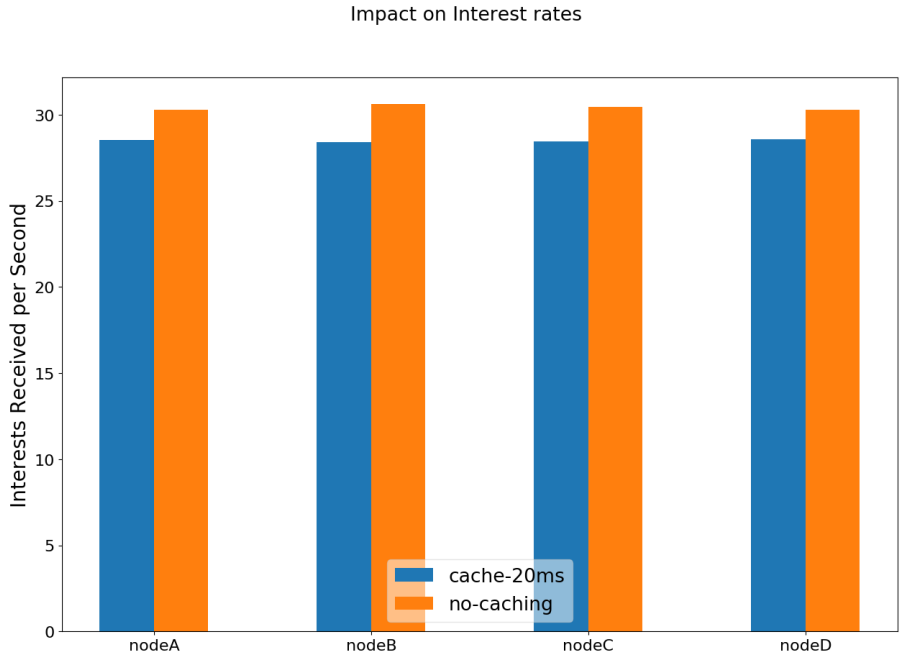

Figure 8.7: Impact of caching on the Interest rates seen by producers in the *tree* topology

These reductions are a direct result of enabling caching. Nodes which fall behind the Interest aggregation period previously resulted in the Interest propogating all the way to the producer. However, by enabling caching, the Interests from nodes which fall behind

are now served by intermediate routers, resulting in a substantial decrease in network traffic.

By enabling caching, the IAF becomes less clear as it is not possible to determine whether the Interests never reached the producer due to caching or Interest aggregation. In either case it is a major benefit to the performance of the game. However, as the same test had previously been performed without caching, a reasonable comparison can be made. The reduction factor in the case of the *dumbbell* topology with caching enabled was 0.33 for all nodes, as seen in figure 8.8.

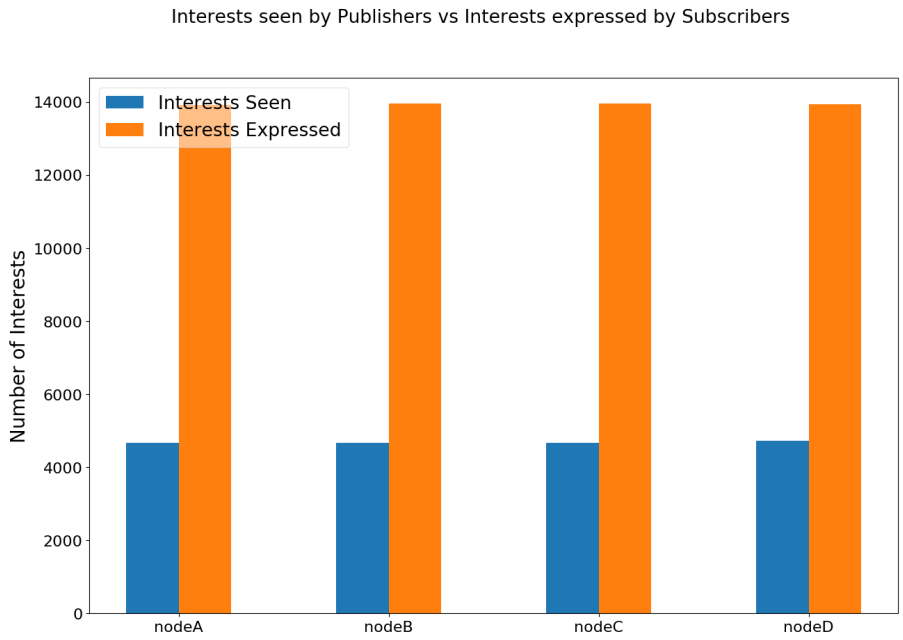

Figure 8.8: Difference between the number of Interests expressed by consumers and seen by producers in the *dumbbell* topology.

The values shown in figure 8.8 can be compared to figure 8.6, which shows the results when caching was disabled. In the case of no caching, the *IAFs* were 0.45, 0.46, 0.45 and 0.44 for nodes A, B, C and D respectively. Thus, enabling caching further improved the reduction factor by between 0.11 and 0.13.

The cache rate of a particular node is heavily dependent on the node's position in the topology. For example, caching essentially only occurs at the intermediate router nodes in *tree* topology. Consider an Interest who's corresponding Data packet has already been produced, this Interest must flow through several Intermediate routers who will all have a cached copy that is **fresher** than the copy at the producer's cache. This is due to the fact that the Data will be cached in the producers CS first and will become non-fresh first. Thus, all intermediate routers will typically be able to serve a cached copy of a Data packet for longer than the producer will.

However, there is an exception to this when the intermediate router's CS becomes full and the Data must be evicted. This will happen more frequently in intermediate routers than in any of the leaf (player) nodes of the *tree* topology as the router nodes must cache Data requested by multiple nodes, whereas the leaf nodes only caches the Data they request. Thus, it is possible than an intermediate router can forward an Interest to a leaf node and have it satisfied by the leaf nodes cache.

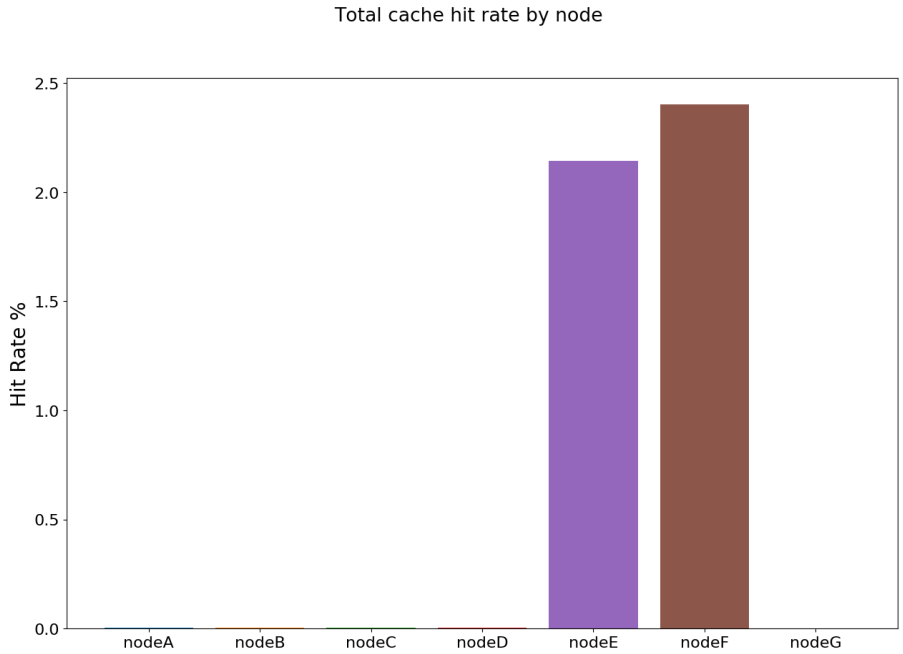

Figure 8.9: Cache rates by node for all nodes in the *tree* topology. Note that the cache rates for nodes A, B, C and D are non zero.

The cache rates shown in figure 8.9 agrees with the expected behaviour previously described. Although these cache rates appear rather low, this is due to the fact that most Interests are aggregated and multicasted, instead of being satisfied by a Data packet in the CS.

Interestingly, node G, the root node, also essentially has a non zero cache rate. Consider the case where A expresses an Interest for a piece of data owned by C (see figure 8.1). This Interest will be forwarded through E, G and F until it reaches C. Once C produces the Data, it is then cached at each hop on the way back. At this point, only B or E will express an Interest for the same piece of data, as A has the data and C is the producer. If B then requests the same packet, the Interest will be satisfied by E, and if D requests the same packet, the Interest will be satisfied by F. Thus, G very rarely serves any cached Data, though it is possible in the case of CS eviction as described previously.

The above results indicate that caching is a major benefit to NDNShooter from a network performance point of view, however the impact it has on the gameplay must also be

considered. In topologies with a large number of hops, it is possible that even relatively small values for *freshness period* can result in players being served extremely old Data packets (see section 9.1).

8.1.3 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, the *dumbbell*, *square* and *tree* topologies performed very similarly. For this reason, only the results for the *dumbbell* topology are shown here and the others can be found in appendix E.

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 4.6. Otherwise, the update is skipped (*skip*). The distribution of the results from the publisher update checks is shown in figure 8.10 for the *dumbbell* topology, and the equivalent figures for the other topologies are given in appendix E.

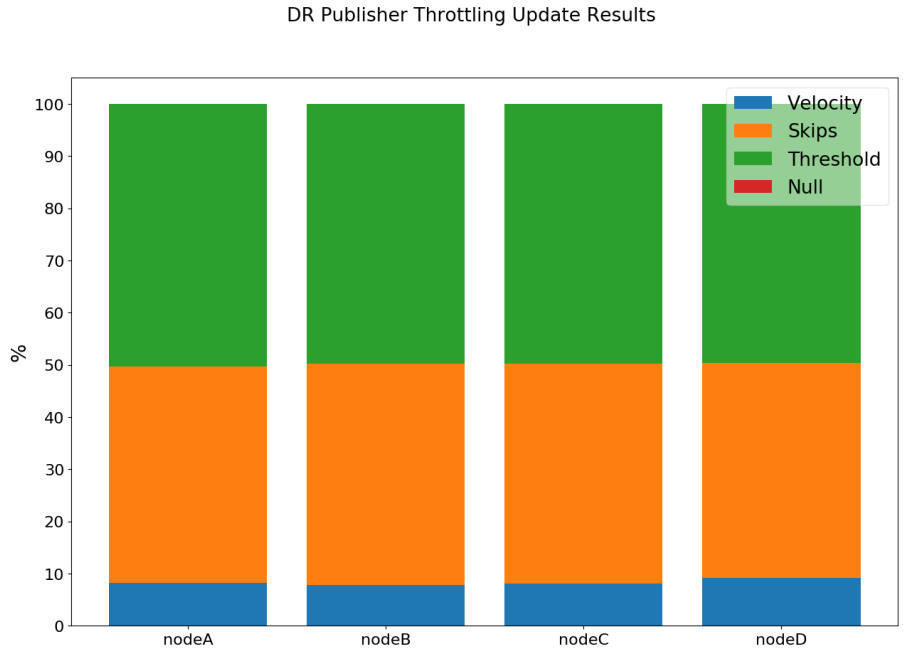

Figure 8.10: Distribution of results from publisher update checks in the *dumbbell* topology. Note there were no *null* updates required in the test for this topology

As seen in figure 8.10, **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 this DR publisher throttling has the potential to hugely impact the ame experience in a negative way as essentially only half of the player updates are now being published. To exmaine 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 ommited.

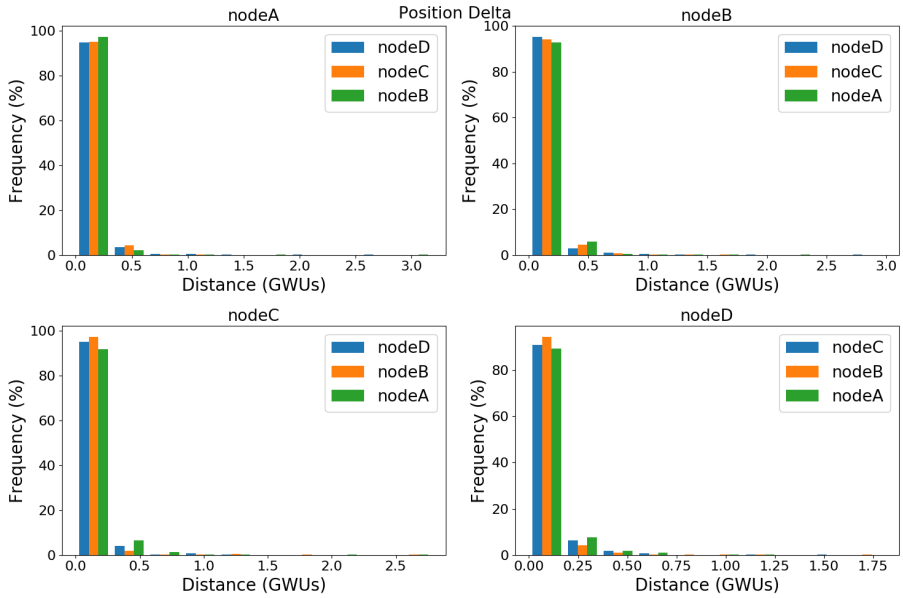

Figure 8.11: *Position deltas* with DR publisher throttling enabled for *dumbbell* topology.

The *position deltas* shown in figure 8.11 are very similar to those shown in figure 8.4, the benchmark test were all optimizations were disabled, indicating that there is very little impact on the game experience when DR publisher throttling is enabled with a threshold of 0.5 GWU.

Unlike with enabling caching, enabling DR publisher throttling does affect the Interest rates in the *linear* topology. As approximately half of the updates were skipped in these tests, the rate of Interests seen by the publishers also dropped accordingly, as seen in figure 8.12 and figure 8.13 for the *dumbbell* and *linear* topologies respectively. The corresponding figures for the *square* and *tree* topologies are provided in appendix E.

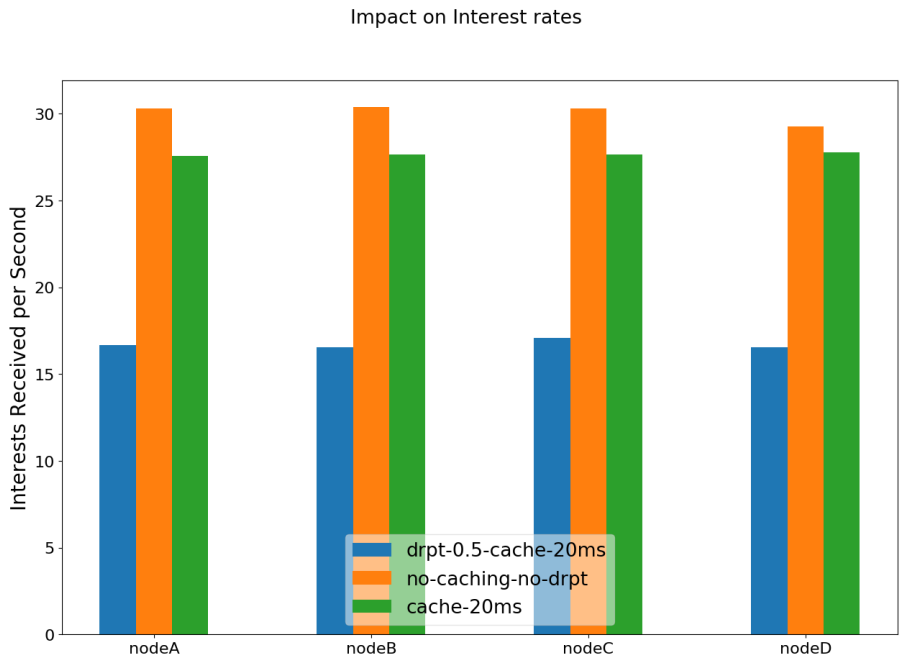

Figure 8.12: Effects of enabling caching with a *freshness period* of 20ms, and DR publisher throttling (*drpt*) with a tolerance of 0.5 GWU on the Interest rate seen by nodes in the *dumbbell* topology.

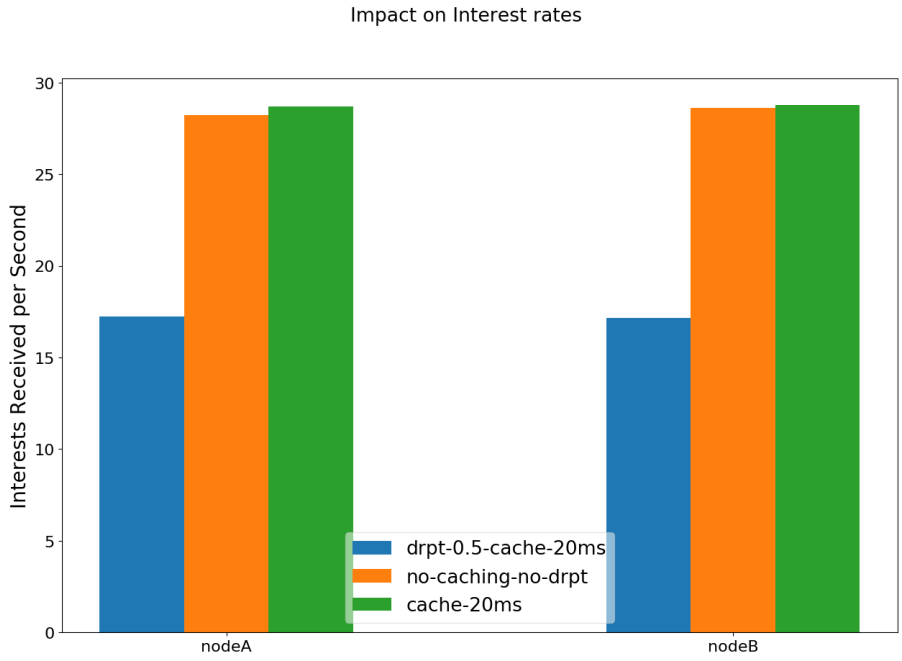

Figure 8.13: Effects of enabling caching with a *freshness period* of 20ms, and DR publisher throttling (*drpt*) with a tolerance of 0.5 GWU on the Interest rate seen by nodes in the *linear* topology.

8.2 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 shwon in figure 8.14.

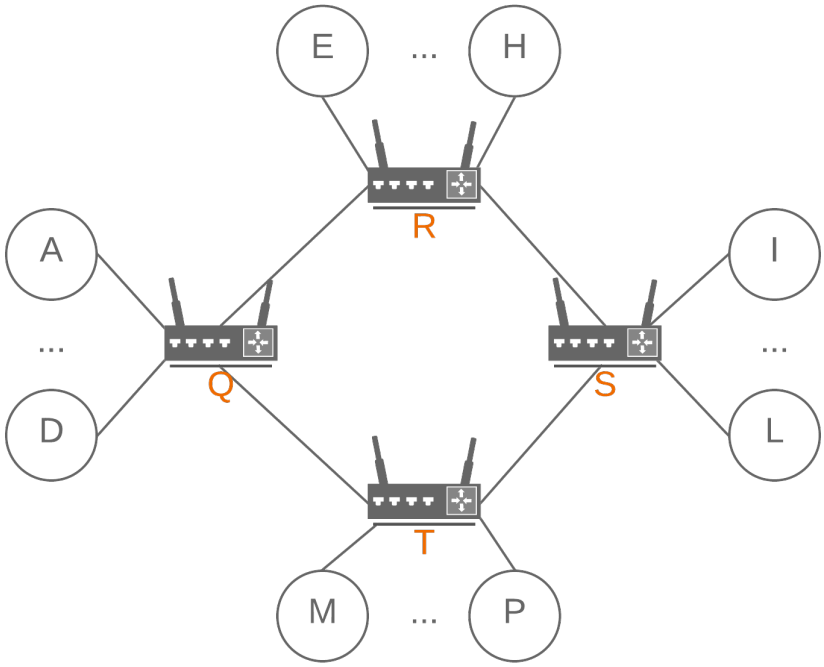

Figure 8.14: The topology used for testing the scalability of NDNShooter, consiting of 4 intermediate routers (Q-T) arranged in a *square* topology, with 4 player nodes behind each router (A-P).

As the topology consists of 20 nodes, initial testing showed that the performance was constrained by the computing power available on the machine used for development. As such, the topology was deployed onto a cluster of 10 Amazon Web Service instances, using a Docker swarm as described in section 7.2.

The game world was also altered to support the larger number of players. The width and height of the game world were doubled to provide space for the increased number of players. Extra impassable boundaries were also added in the form of squares in each of the four corners of the game world. This was done to encourage hot spotting which is a common issue in MOGs. The adjusted game world is shown in 8.15.

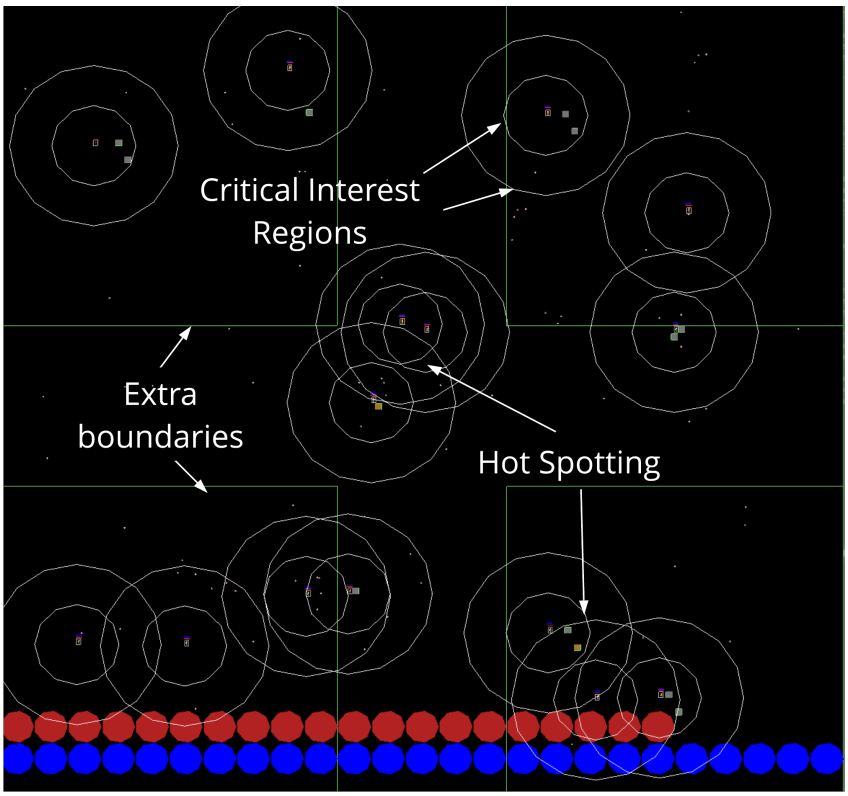

Figure 8.15: A screenshot of the adjusted game world during scalability testing. The screenshot shows the extra boundaries added to the game world, the occurence of hot spots in the game world and the critical interest regions defined by r_{full} and r_{min} used by the Interest management system.

8.2.1 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 8.16 and figure 8.17 respectively and the effect of Interest aggregation is shown in figure 8.18.

As seen in the figures above, NDNShooter scales up to 16 players very well and the results are comparable to those seen in earlier tests with 2 or 4 players.

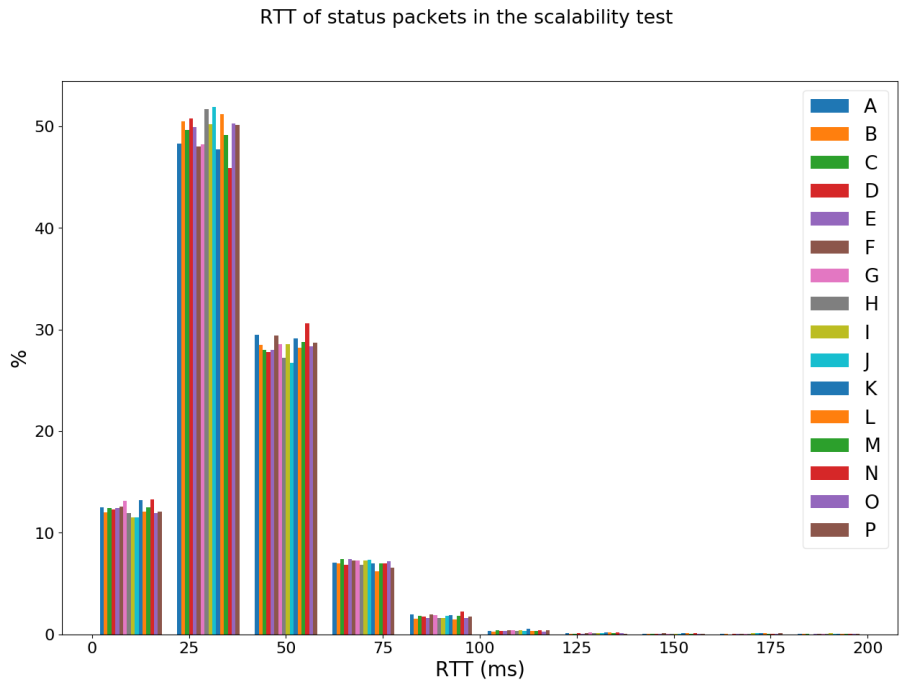

Figure 8.16: *RTTs* for each of the nodes in the scalability test with no IM or DRPT. As before, there is a peak around 33ms which corresponds to the publisher update frequency of 30Hz

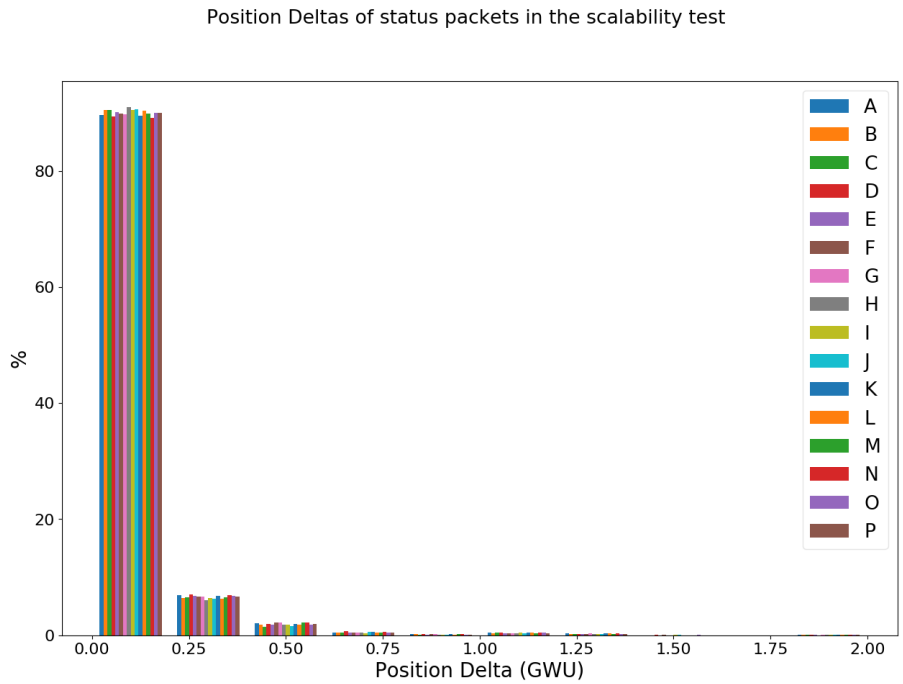

Figure 8.17: *Position Deltas* for each of the nodes in the scalability test with no IM or DRPT. As before, the values are very close to zero in most cases indicating the players' positions are very tightly syncrhonzied, even with 16 players.

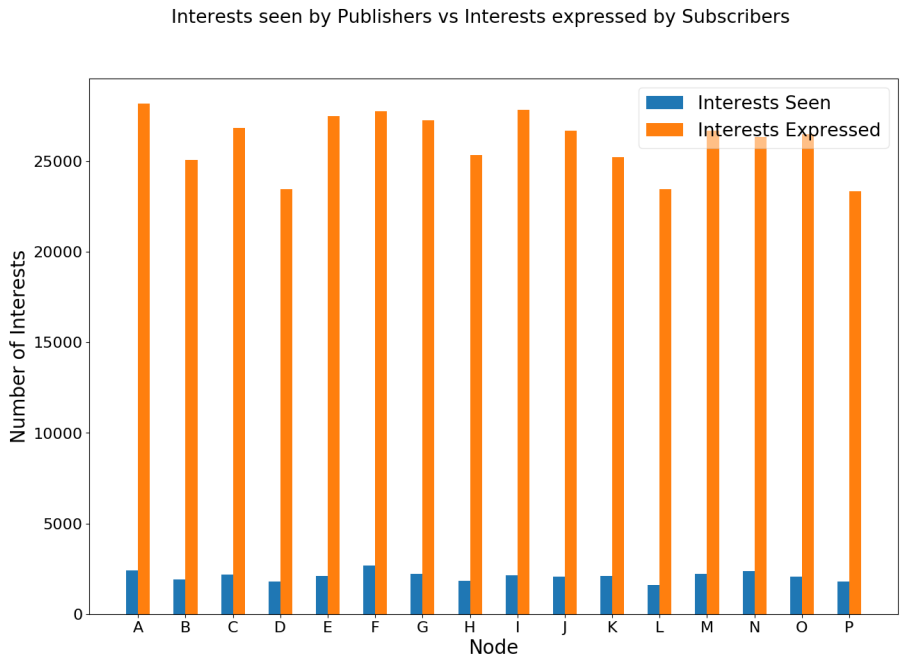

Figure 8.18: The effect of Interest aggregation in the scalability test with no IM and no DRPT. The *Interest aggregation factor (IAF)* ranged between 0.07 and 0.09 in this case, which is significantly lower than the best *IAF* seen in previous tests were optimizations were enabled. However, *IAFs* are expected to decrease as more players are added, due to the overall larger number of Interests expressed.

8.2.2 Enabling Dead Reckoning Publisher Throttling (DRPT)

With a benchmark established, *DRPT* was then re-enabled with the same values used in previous tests. As *DRPT* effectively slows down the publisher update rate, the expected effect of enabling DRPT is an increase in RTTs and a decrease in the rate of Interests seen by producers.

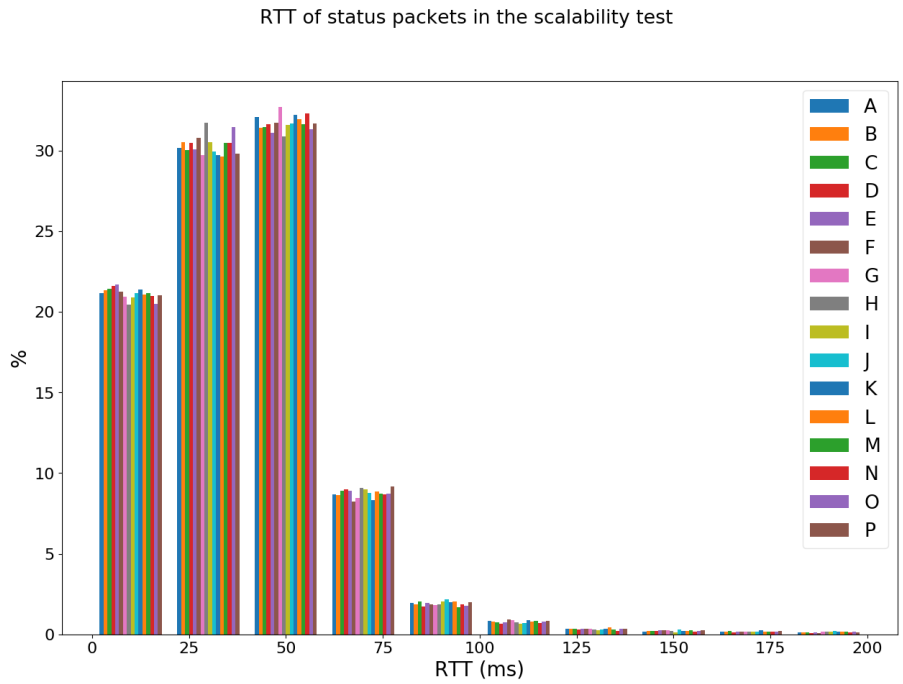

Figure 8.19: *RTTs* for each of the nodes in the scalability test with DRPT enabled and IM disabled. The peak increases from the previous value of 33ms due to the effective decrease in the rate of publsiher updates, caused by DRPT.

As seen in previous tests, enabling *DRPT* caused a substantial drop in the rate of Interests seen by producers.

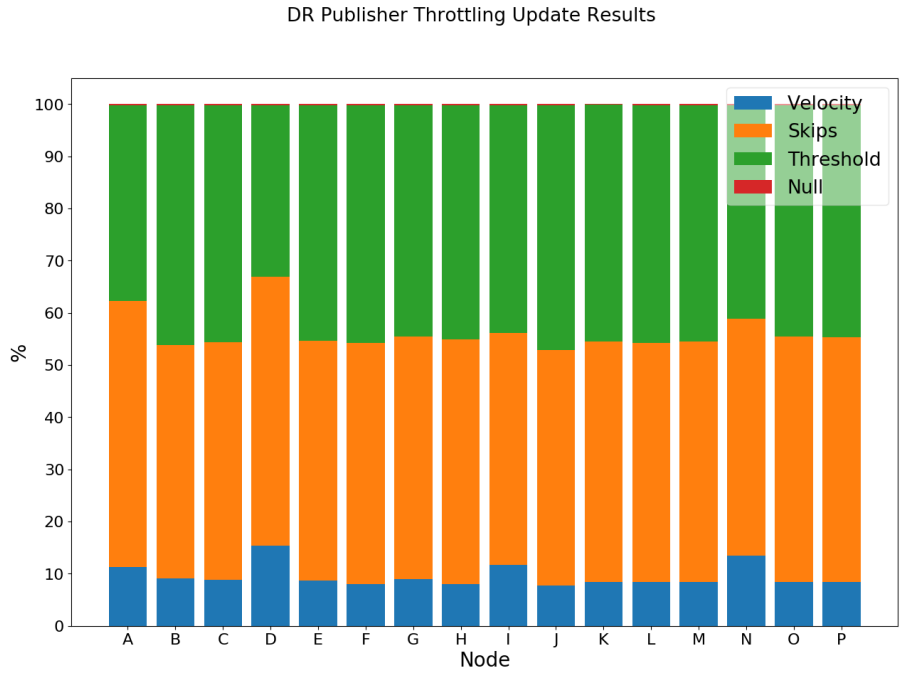

Figure 8.20: The distribution of the results of publisher update checks for each of the players in the *scalability test*. Approximately 40-50% of the updates could be skipped as a result of *DRPT*.

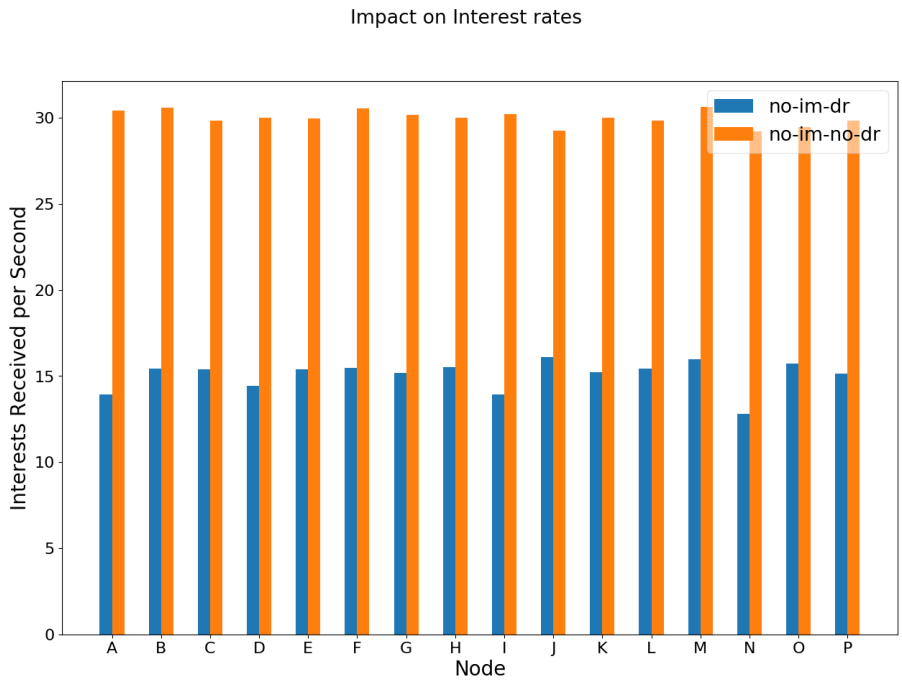

Figure 8.21: The reduction in Interest rates seen by producers in the scalablity test as a result of enabling *DRPT*. The figure shows a reduction of approximately 40-50% which aligns with the skipped updates seen in figure 8.20

8.2.3 Enabling Interest Management (IM)

IM enables players to intelligently control the rate at which they express Interest for remote objects, based on the distance the object is from the player. *DRPT* was once again disabled to examine the impact of IM alone. IM was enabled using the following parameters, which were defined in section 4.7:

$$r_{full} = 10\ GWU$$
$$r_{min} = 20\ GWU$$
$$MAX\_SLEEP = 2\ seconds$$

There are two expected outcomes of enabling IM. The first is a significant reduction in the rate of Interests seen by publishers, due to the reduced rate of Interests expressed by consumers. The second is a likely increase in *position deltas* due to the longer time between updates, during which remote players may have changed direction. The results of the test agree with these two expected outcomes and can be seen in figure 8.17 and figure 8.23 respectively.

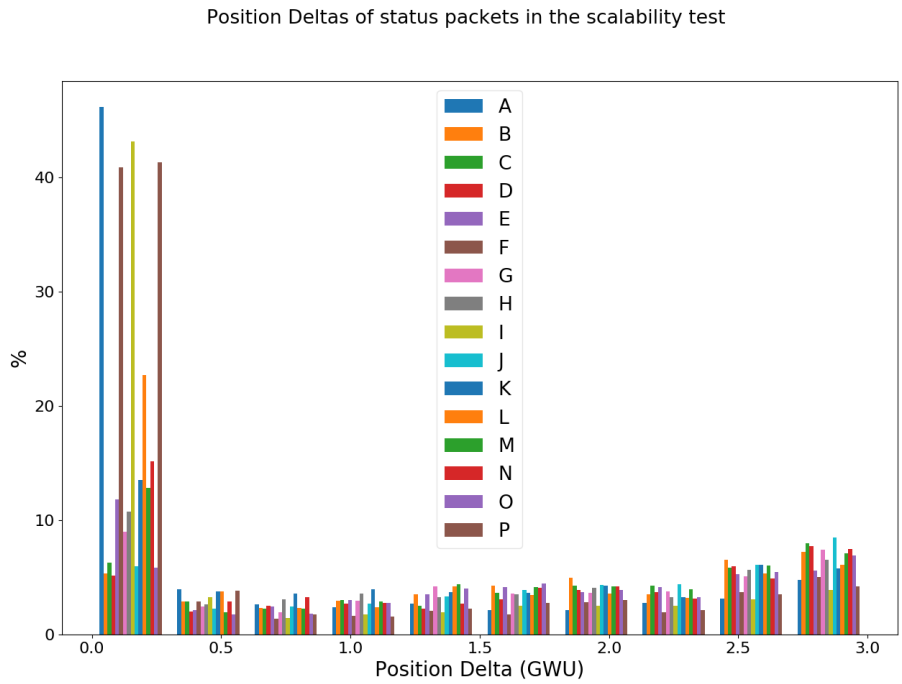

Figure 8.22: The substantial increase in *position deltas* as a result of enabling IM. This is due to consumers reducing the rate at which they express Interests for updates to remote objects which are far away.

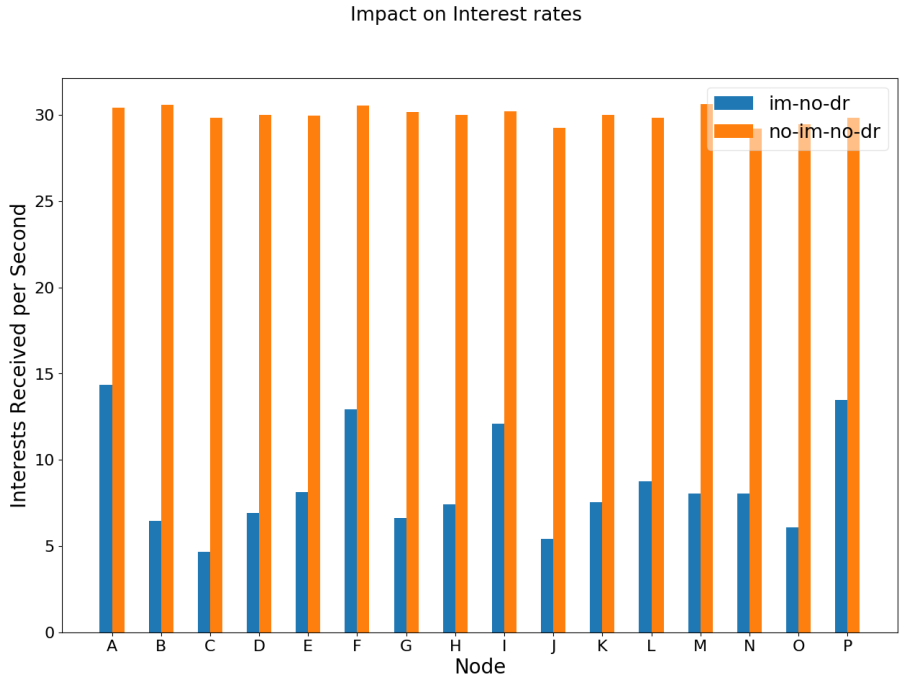

Figure 8.23: The reduction in Interest rates seen by producers in the scalablity test as a result of enabling *IM*.

8.2.4 Enabling DRPT and IM

Finally, the test was repeated a fourth time to examine the impacts of enabling both DRPT and IM. 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 8.24 agree with this threory.

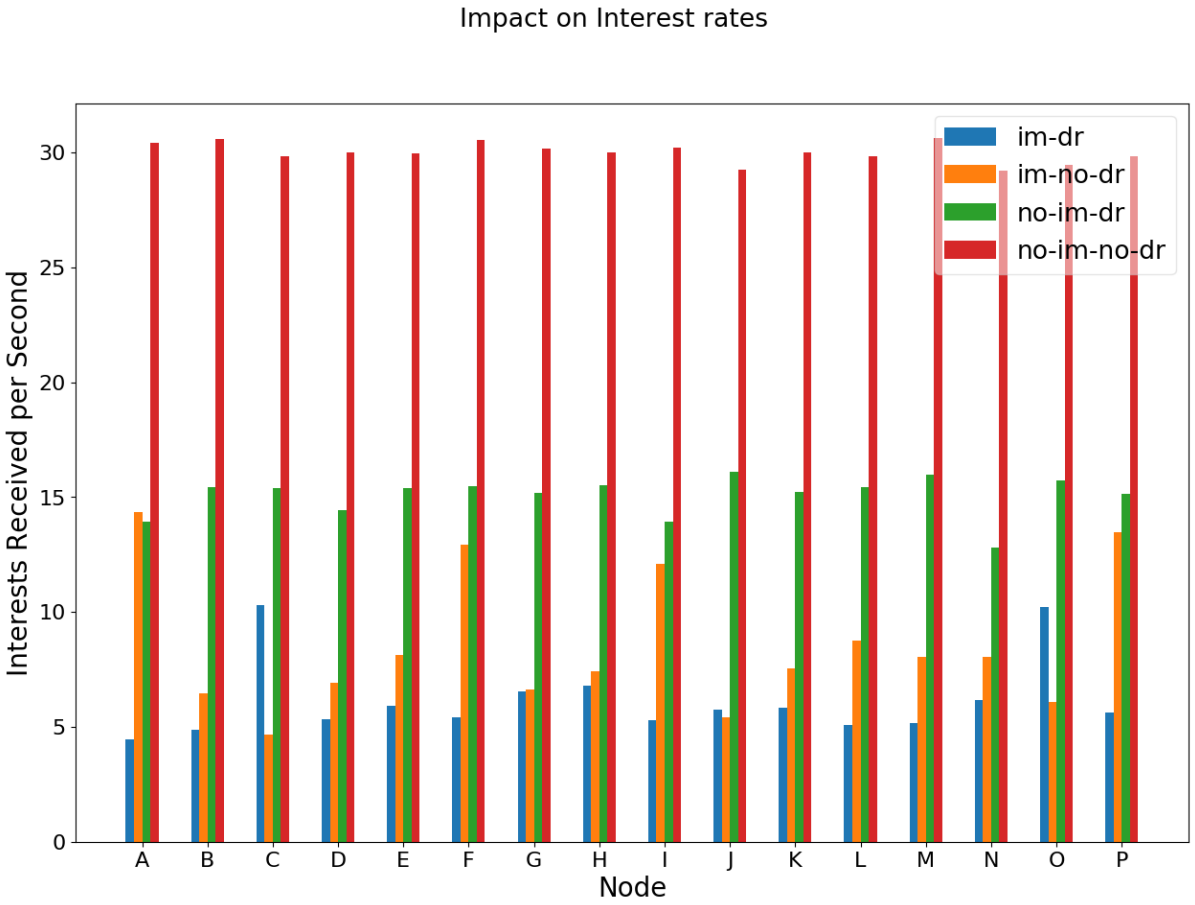

Figure 8.24: The impacts of enabling IM, DR and both IM and DR on the Interest rates seen by in the scalability tests.

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 8.24 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.

9 Conclusion

The performance of NDNShooter suggests that NDN is an excellent candiate for MOG communications. The variety of the data produced by MOGs requires several syncrhonization strategies for optimum performance. Existing solutions for dataset syncrhonization 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 syncrhonize 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 syncrhonization protocols, which lead to the development of the novel syncrhonization protocol used in NDNShooter.

The testing performed in section 8.1 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. Crticially, 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 noticable 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 syncrhonization, meaning producers need not store state for each consumer, which provides high consumer scalability.

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.

9.1 Future Work

Due to the time limitations associated with this project, several features and research areas were ommited. 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:

9.1.1 IP Based Backend Module

As the backend module used in NDNShooter is entirely modular, an IP based backend module could be developed to faciliatate a direct comparison between an IP based solution and the NDN solution developed during this research.

9.1.2 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 inceasing the amount of caching possible in NDNShooter.

9.1.3 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 syncrhonization 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.

9.1.4 Build a More Rhobust Interaction API

The limitations of the Interaction API used in NDNShooter were discussed in section 4.5.2. Adding functionality to the Interaction API to support more complex game mechanics would go a long way in bringing NDNShooter up to the standard of real MOGs played today.

9.1.5 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 analyzed to gain a deeper understanding of how the game scales. As shown in section 8.2, NDNShooter could easily support 16 players without causing any noticable impact on game performance. As such, larger scalability tests are required to determine the upper limit to the scalability of NDNShooter.

Accessed on for bib

Bibliography

[1] Ndn executive summary, . URL `https://named-data.net/project/execsummary/`.

[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. ISSN 00010782. doi: 10.1145/2063176.2063204.

[3] Ndn packet specification, . URL `https://named-data.net/doc/NDN-packet-spec/current/`.

[4] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Patrick Crowley, Christos Papadopoulos, Lan Wang, Beichuan Zhang, et al. Named data networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

[5] Xiaoke Jiang, Jun Bi, and You Wang. What benefits does ndn have in supporting mobility. In *2014 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2014.

[6] Alexander Afanasyev, Xiaoke Jiang, Yingdi Yu, Jiewen Tan, Yumin Xia, Allison Mankin, and Lixia Zhang. Ndns: A dns-like name service for ndn. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

[7] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D Thornton, Diana K Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, and Christos Papadopoulos. Named data networking project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, 157:158, 2010.

[8] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. A case for stateful forwarding plane. *Computer Communications*, 36 (7):779–791, 2013.

[9] Cesar Ghali, Gene Tsudik, Ersin Uzun, and Christopher A Wood. Living in a pitless world: A case against stateful forwarding in content-centric networking. *arXiv preprint arXiv:1512.07755*, 2015.

[10] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. In *ACM SIGCOMM Computer Communication Review*, volume 15, pages 44–53. ACM, 1985.

[11] Ndn repo-ng github page, . URL `https://github.com/named-data/repo-ng`.

[12] Ndn repo-ng homepage, . URL `https://redmine.named-data.net/projects/repo-ng/wiki`.

[13] ndnsim forwarding strategies, . URL `https://ndnsim.net/2.1/fw.html`.

[14] Hila Ben Abraham and Patrick Crowley. Forwarding strategies for applications in named data networking. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, pages 111–112. ACM, 2016.

[15] E. Rescorla T. Dierks. Rfc 5246, the transport layer security (tls) protocol version 1.2. URL `https://tools.ietf.org/html/rfc5246`.

[16] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. An overview of security support in named data networking. *IEEE Communications Magazine*, 56(11):62–68, 2018. ISSN 0163-6804.

[17] Nfd github page, . URL `https://github.com/named-data/NFD`.

[18] Thomas Clausen and Philippe Jacquet. Optimized link state routing protocol (olsr). Technical report, 2003.

[19] EW Djikstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[20] Olivier Bonaventure. Link state routing. URL `http://cnp3book.info.ucl.ac.be/principles/linkstate.html`.

[21] 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.

[22] Ndn common client libraries (ndn-ccl) documentation, . URL `https://named-data.net/codebase/platform/ndn-ccl/`.

[23] Ndn tools github page, . URL https://github.com/named-data/ndn-tools.

[24] ndnsim ns-3 based named data networking simulator, . URL http://ndnsim.net/current.

[25] Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. On the evolution of ndnsim: An open-source simulator for ndn experimentation. *ACM SIGCOMM Computer Communication Review*, 47(3):19–33, 2017.

[26] Mini-ndn github page, . URL https://github.com/named-data/mini-ndn.

[27] Mininet homepage, . URL http://mininet.org/.

[28] Van Jacobson, Diana K Smetters, Nicholas H Briggs, Michael F Plass, Paul Stewart, James D Thornton, and Rebecca L Braynard. Voccn: voice-over content-centric networks. In *Proceedings of the 2009 workshop on Re-architecting the internet*, pages 1–6. ACM, 2009.

[29] Bur Goode. Voice over internet protocol (voip). *Proceedings of the IEEE*, 90(9): 1495–1517, 2002.

[30] Peter Gusev, Zhehao Wang, Jeff Burke, Lixia Zhang, Takahiro Yoneda, Ryota Ohnishi, and Eiichi Muramoto. Real-time streaming data delivery over named data networking. *IEICE Transactions on Communications*, 99(5):974–991, 2016.

[31] Peter Gusev and Jeff Burke. Ndn-rtc: Real-time videoconferencing over named data networking. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*, pages 117–126. ACM, 2015.

[32] Alexander Afanasyev Spyridon Mastorakis, Peter Gusev and Lixia Zhang. Real-time data retrieval in named data networking. In *Proceedings of IEEE International Conference on Hot Information-Centric Networking (HotICN'2018)*, August 2018. URL https://named-data.net/publications/hoticn18realtime-retrieval.

[33] Shang Wentao, Yu Yingdi, Wang Lijing, Afanasyev Alexander, and Zhang Lixia. A survey of distributed dataset synchronization in named data networking. Report, 2017. URL https://named-data.net/wp-content/uploads/2017/05/ndn-0053-1-sync-survey.pdf.

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

[35] isync: A high performance and scalable data synchronization protocol for named data networking, September 2014. URL https://named-data.net/publications/poster_isync/.

[36] 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.

[37] 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.

[38] 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.

[39] spirosmastorakis. Roundsync ndnsim github page. URL `https://github.com/spirosmastorakis/RoundSync`.

[40] Valve Corporation. Source engine multiplayer networking. URL `https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`.

[41] Unity game development platform homepage. URL `https://unity.com/`.

[42] Libgdx homepage. URL `https://libgdx.badlogicgames.com/`.

[43] 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. ISSN 03600300. doi: 10.1145/2522968.2522977. URL `http://elib.tcd.ie/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=91956728`.

[44] Ta Nguyen Binh Duong and Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. In *The 11th IEEE International Conference on Networks, 2003. ICON2003.*, pages 131–136. IEEE, 2003.

[45] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 48. ACM, 2006.

[46] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[47] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[48] 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.

[49] Declan Delaney, Seamus McLoone, and Tomas Ward. A novel convergence algorithm for the hybrid strategy model packet reduction technique. 2005.

[50] Alan Kenny, Séamus McLoone, Tomás Ward, and Declan Delaney. Using user perception to determine suitable error thresholds for dead reckoning in distributed interactive applications. 2006.

[51] Jean-Sébastien Boulanger. *Interest management for massively multiplayer games*. PhD thesis, McGill University, 2006.

[52] César Cañas, Kaiwen Zhang, Bettina Kemme, Jörg Kienzle, and Hans-Arno Jacobsen. Publish/subscribe network designs for multiplayer games. In *Proceedings of the 15th International Middleware Conference*, pages 241–252. ACM, 2014.

[53] Zening Qu and Jeff Burke. Egal car: A peer-to-peer car racing game synchronized over named data networking. 2012. URL `https://named-data.net/wp-content/uploads/TRegalcar.pdf`.

[54] 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. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-84942310686&partnerID=40&md5=6e1558c28e5b090b0ea43690d2ba50dd`.

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

[56] Box2d physics engine homepage. URL `https://box2d.org/`.

[57] Libgdx ashley github page. URL `https://github.com/libgdx/ashley`.

[58] Google's protocol buffers homepage. URL `https://developers.google.com/protocol-buffers/`.

[59] Nfd control documentation, . URL `https://named-data.net/doc/NFD/0.1.0/manpages/nfdc.html`.

[60] Nfd best-route strategy v2 issue, . URL `https://redmine.named-data.net/issues/1871`.

[61] Ndn ccl docs: Blob, . URL https://named-data.net/doc/ndn-ccl-api/blob.html.

[62] Joshua Bloch. *Effective java (the java series)*. Prentice Hall PTR, 2008.

[63] Google's guice github page. URL https://github.com/google/guice.

[64] What is a container? - docker documentation, . URL https://www.docker.com/resources/what-container.

[65] Peter gusev's docker image for an ndn node, . URL https://github.com/peetonn/ndn-docker/tree/master/node.

[66] Docker compose documentation, . URL https://docs.docker.com/compose/.

[67] Dropwizard metrics homepage. URL https://metrics.dropwizard.io/4.0.0/.

Appendices

App. A Invertible Bloom Filters

IBFs are an extension to standard BFs which replace the simple bit array used in BFs with a list of objects. IBFs extend BFs to support element retrieval and deletion. The indices produced by the hash functions are used as indices into this list in order to extract the objects of interest for a given element. The objects in the IBF list contain a *key*, a *value* and a *count*. IBF operations are defined as follows, where an o refers to an object at index i such that i is the output of hashing the element with one of the hash functions:

insert(key, val) For each o, $o.key := o.key\ XOR\ key$, the new value becomes $o.value := o.value\ XOR\ val$, $o.count{+}{+}$.

delete(key) Assuming the element had been inserted, for each o, the new key becomes $existingKey\ XOR\ deleteKey$, the new value becomes $existingValue\ XOR\ deleteValue$ and the count is decremented.

get(key) There are three cases to consider when retrieving an *value* by *key*:

- If the *count* of **any** of the objects of interest are zero, the element was never inserted.
- If none of the objects of interest have a $count == 1$, the element cannot be retrieved but may have been inserted.
- If any of the objects of interest have a *key* which matches the *key* to be retrieved, then the *value* of that object is returned. Otherwise, the element was never inserted.

App. B Docker Compose file for players A and B

The most basic topology is two players who are directly connected to one and other and this is shown in figure B.1.

Figure B.1: Players A and B directly connected via a single link

A simplified version of the resulting Docker Compose file generated for testing this topolgoy is shown in code listing B.1. Of particular interest are the *enviroment* and *network* fields for each of the two services.

The *environment* field specifies the location of the NLSR config file that the node should use and the command the node should use to start NDNShooter with the appropriate parameters.

The *network* field specifies the name of the overlay network to connect to and the alias that this service should be given on that overlay network. This allows the node to be accessible at this alias by other nodes in the overlay network.

```
1 version: '3.3'
2
3 services:
4   A:
5     image: stefanolupo/ndn:ndn-node
6     environment:
7       - NLSR_CONFIG=/NLSR/topologies/linear/nodeA-nlsr.conf
8       - GAME=java -jar /NdnGame/NdnGameLibGdxDesktop-1.0-SNAPSHOT.jar -a ws -hl -n
            nodeA
```

```
9       networks:
10        private-ndn-overlay:
11          aliases:
12            - nodea.ndngame.com
13      deploy:
14        mode: replicated
15        replicas: 1
16        endpoint_mode: dnsrr
17    B:
18      image: stefanolupo/ndn:ndn-node
19      environment:
20        - NLSR_CONFIG=/NLSR/topologies/linear/nodeB-nlsr.conf
21        - GAME=java -jar /NdnGame/NdnGameLibGdxDesktop-1.0-SNAPSHOT.jar -a ws -hl -n
              nodeB
22      networks:
23        private-ndn-overlay:
24          aliases:
25            - nodeb.ndngame.com
26      deploy:
27        mode: replicated
28        replicas: 1
29        endpoint_mode: dnsrr
30
31  networks:
32    private-ndn-overlay:
33      driver: overlay
```

Code Listing B.1: Docker Compose file for a two player topology

App. C Figures For Other Topologies Using No Optimizations

C.1 Round Trip Times

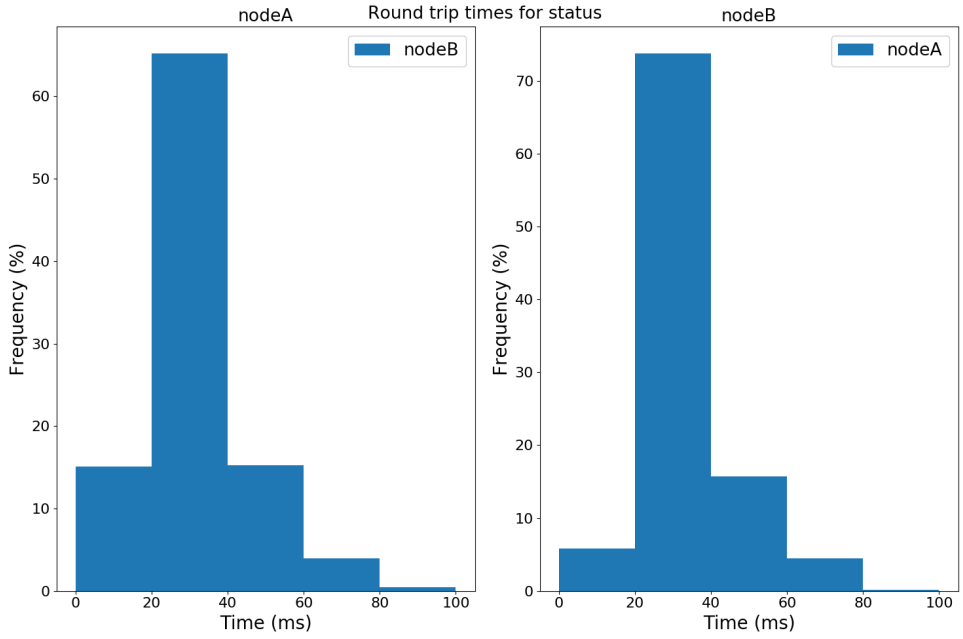

Figure C.1: *RTTs* for *PlayerStatus* in the *linear* topology

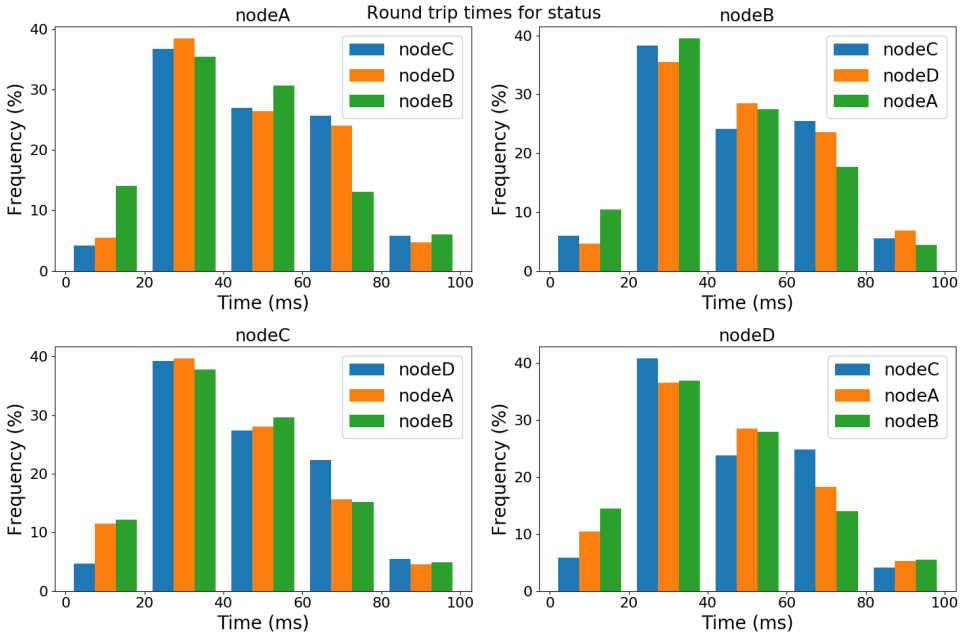

Figure C.2: *RTTs* for *PlayerStatus* in the *square* topology

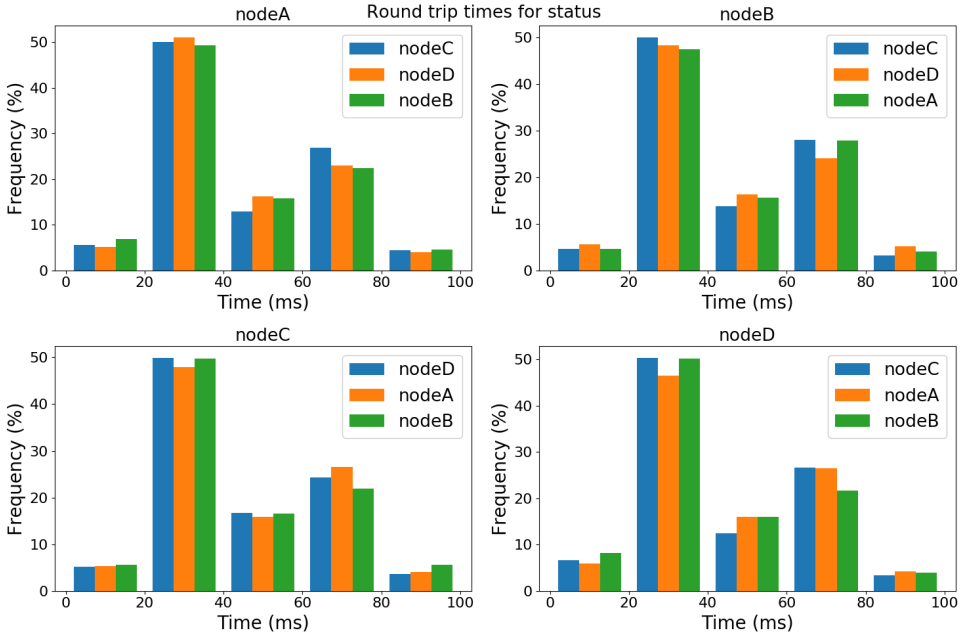

Figure C.3: *RTTs* for *PlayerStatus* in the *tree* topology

C.2 Position Deltas

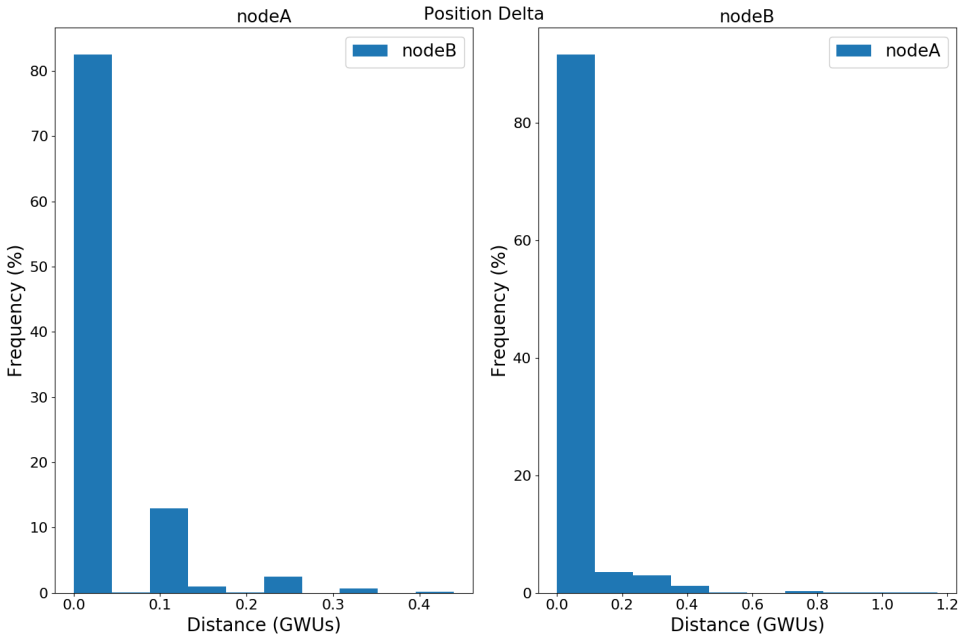

Figure C.4: *Position deltas* in the *linear* topology

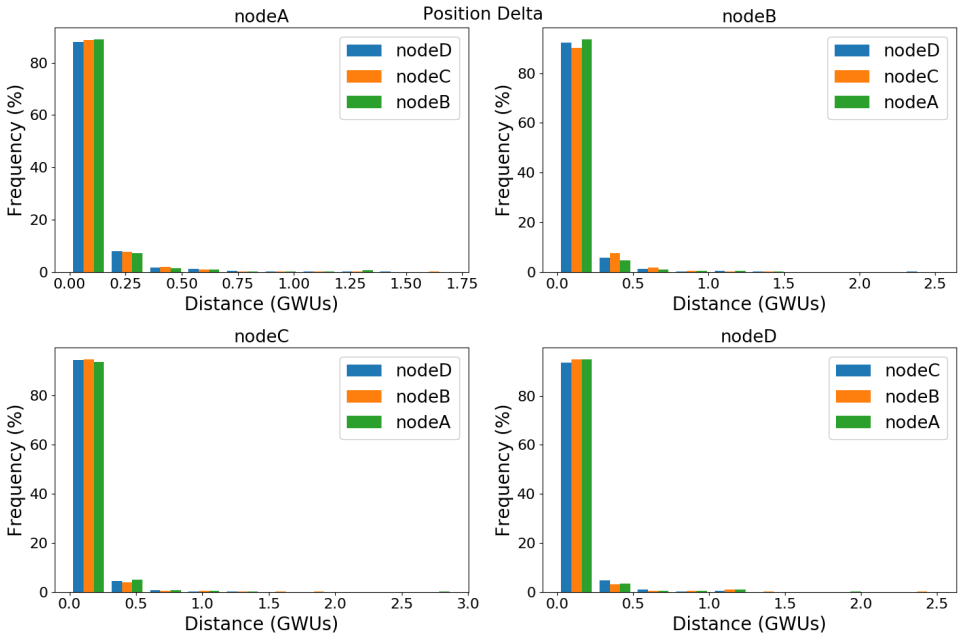

Figure C.5: *Position deltas* in the *square* topology

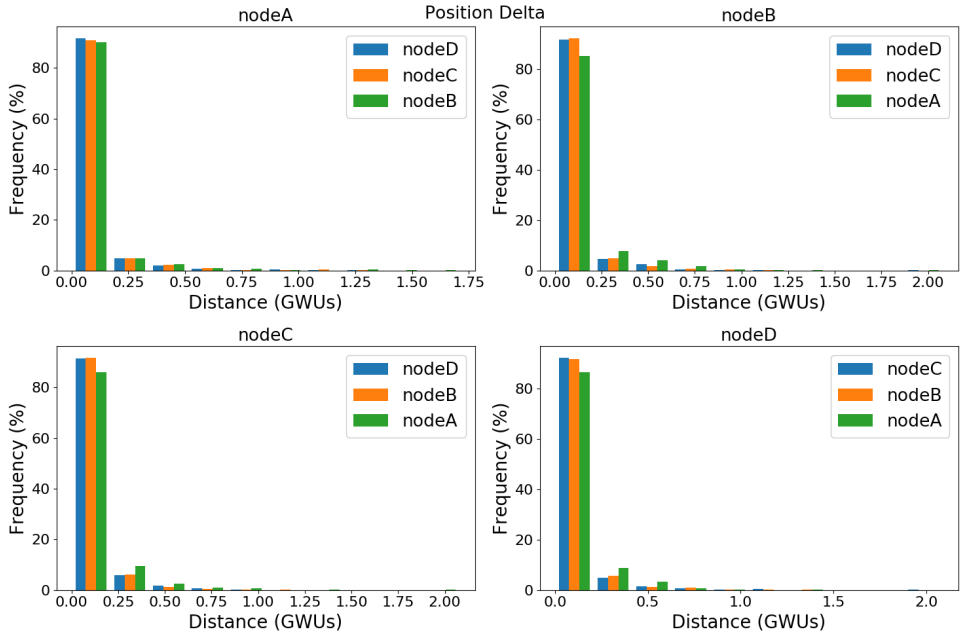

Figure C.6: *Position deltas* in the *tree* topology

C.3 Interest Aggregation

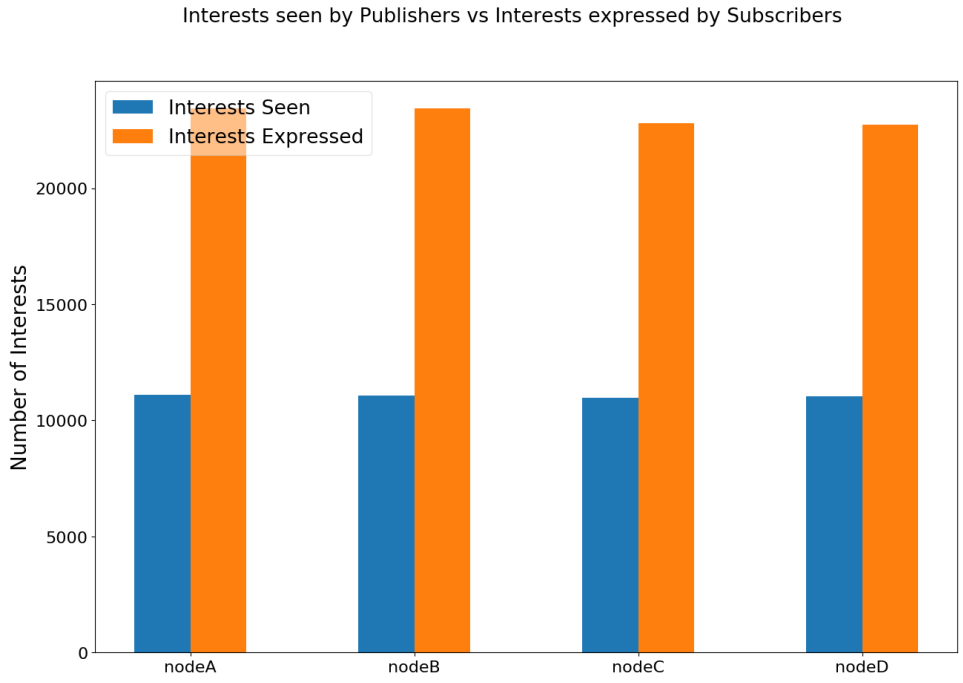

Figure C.7: Interest aggregation in the *square* topology. Nodes A, B, C and D had IAFs of 0.47, 0.47, 0.48 and 0.48 respectively.

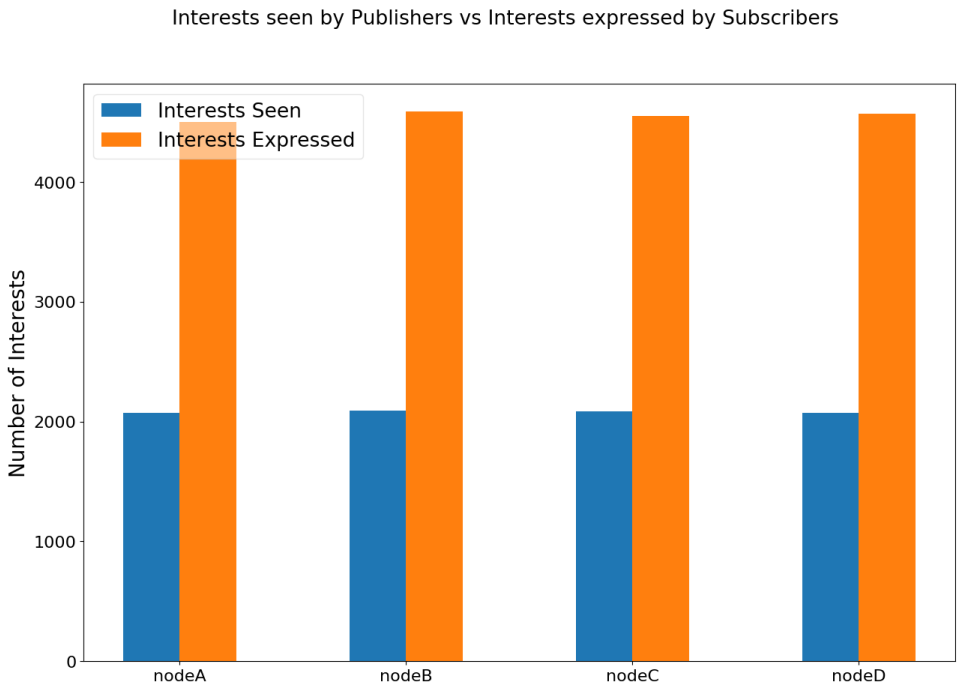

Figure C.8: Interest aggregation in the *tree* topology. Nodes A, B, C and D had IAFs of 0.46, 0.45, 0.45 and 0.45 respectively.

App. D Figures For Other Topologies with Caching Enabled

D.1 Impact of Enabling Caching on Interest Rates

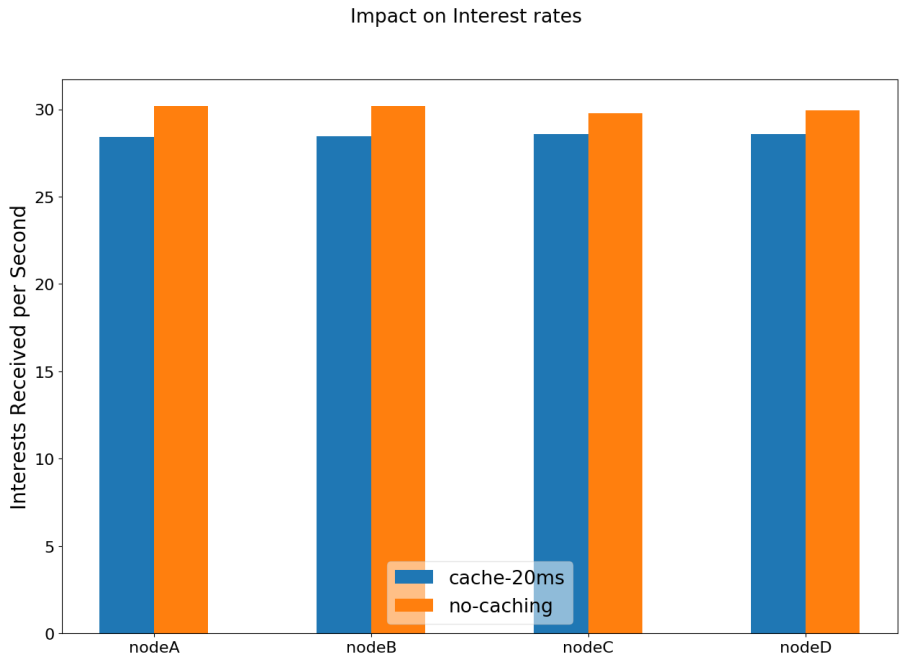

Figure D.1: Impact of caching on the Interest rates seen by producers in the *square* topology

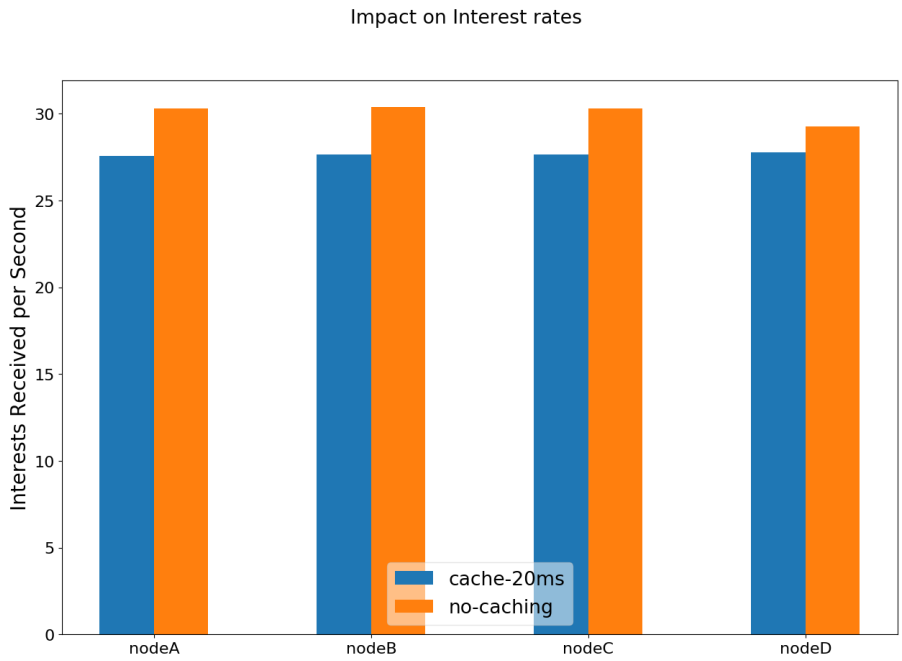

Figure D.2: Impact of caching on the Interest rates seen by producers in the *dumbbell* topology

D.2 Interest Reduction Factor due to Enabling Caching

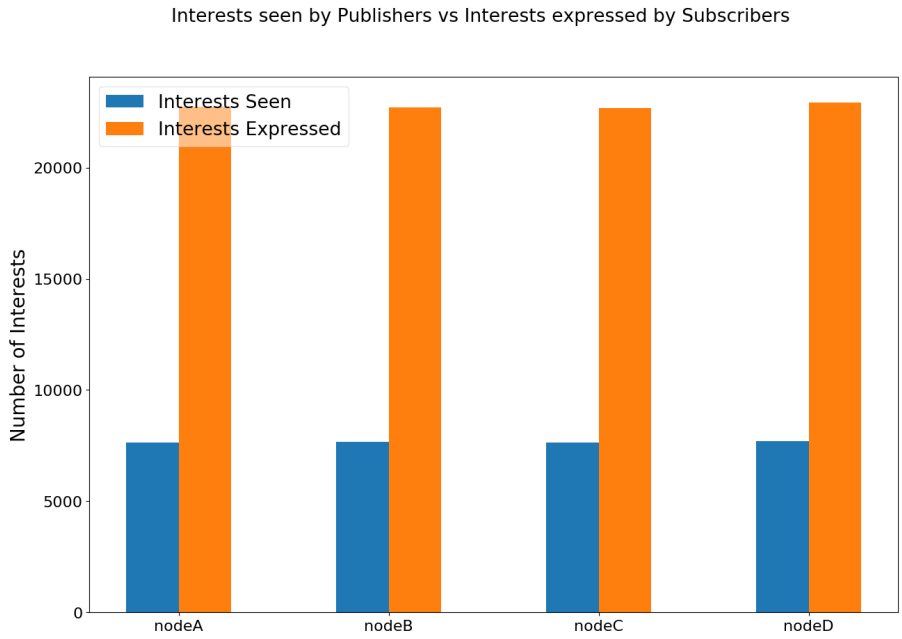

Figure D.3: Difference between the number of Interests expressed by consumers and seen by producers in the *square* topology.

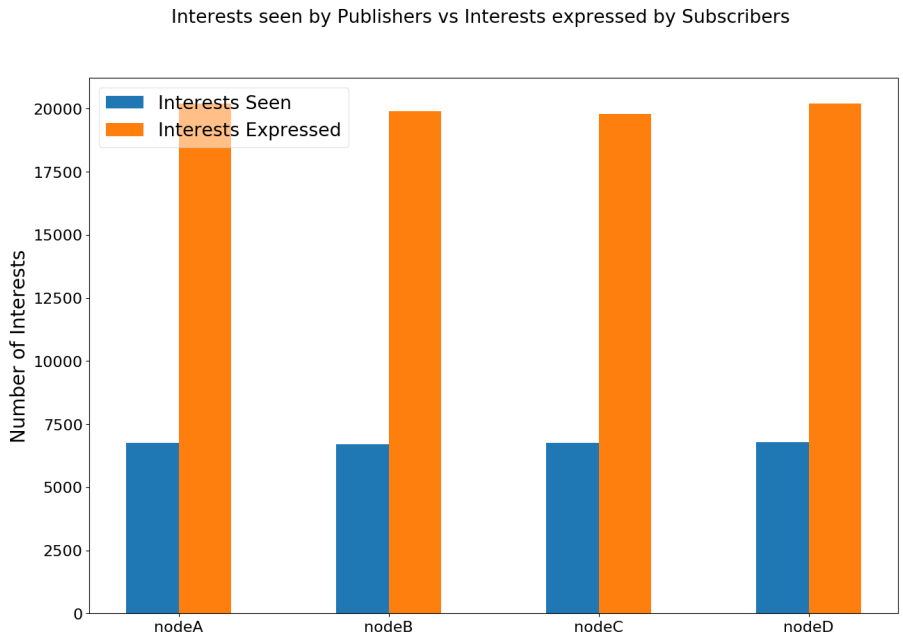

Figure D.4: Difference between the number of Interests expressed by consumers and seen by producers in the *tree* topology.

App. E Figures For Other Topologies with DR Publisher Throttling Enabled

E.1 Publisher Update Results

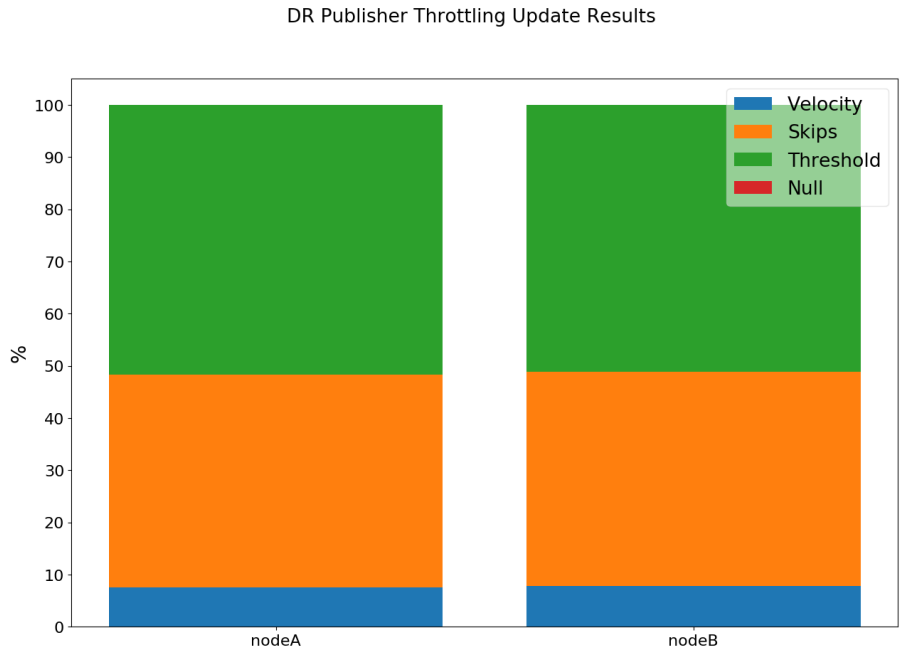

Figure E.1: Distribution of results from publisher update checks in the *linear* topology.

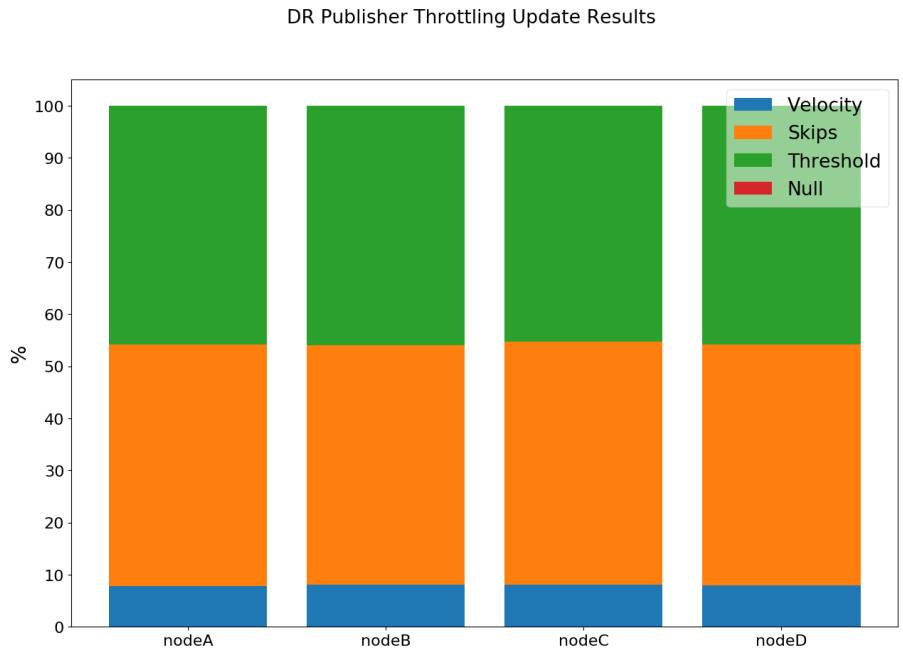

Figure E.2: Distribution of results from publisher update checks in the *square* topology.

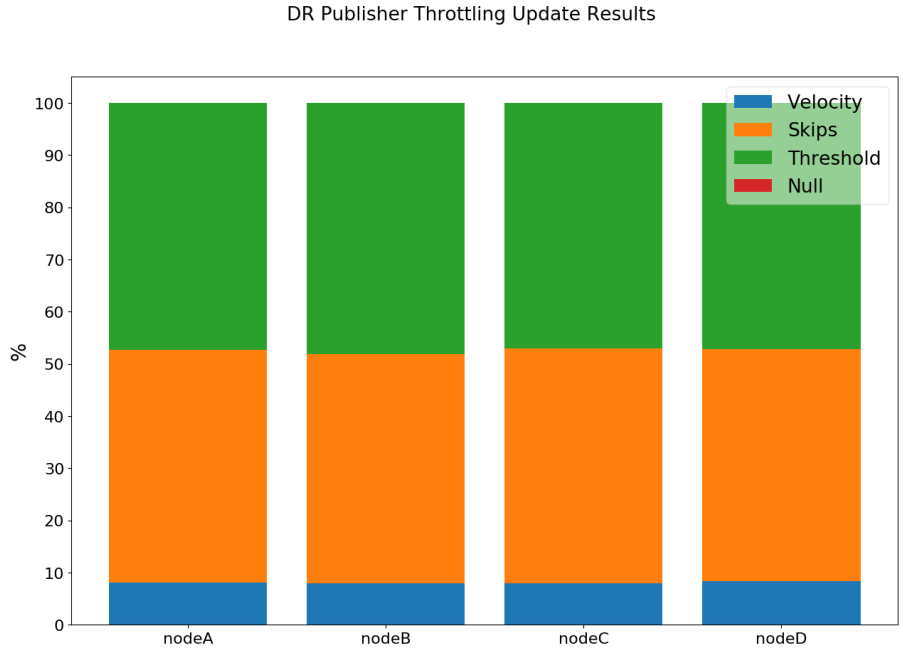

Figure E.3: Distribution of results from publisher update checks in the *tree* topology.

E.2 Effects of Enabling Caching and DR Publisher Throttling

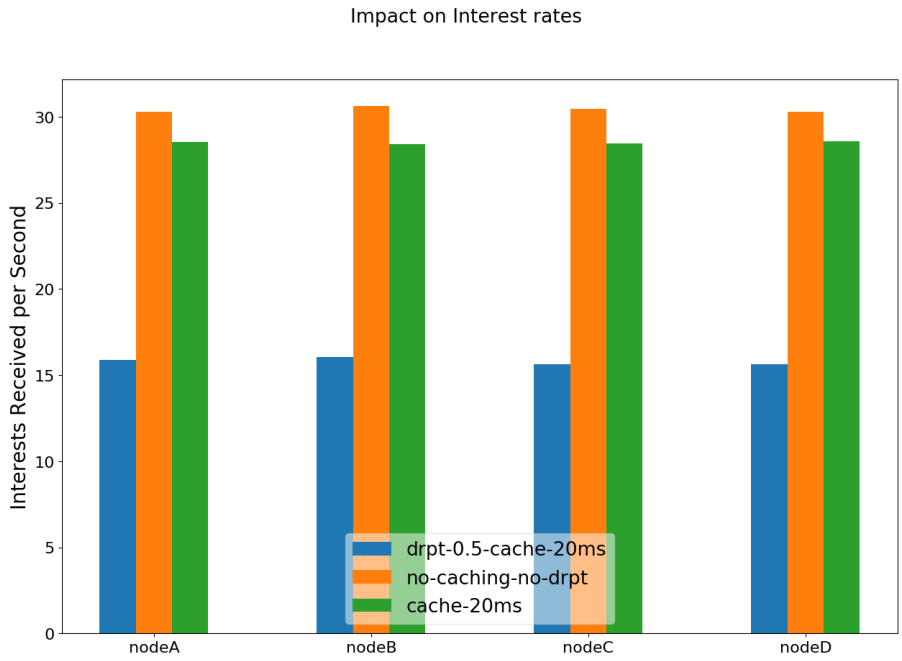

Figure E.4: Effects of enabling caching and DR publisher throttling on the Interest rate seen by nodes in the *tree* topology.

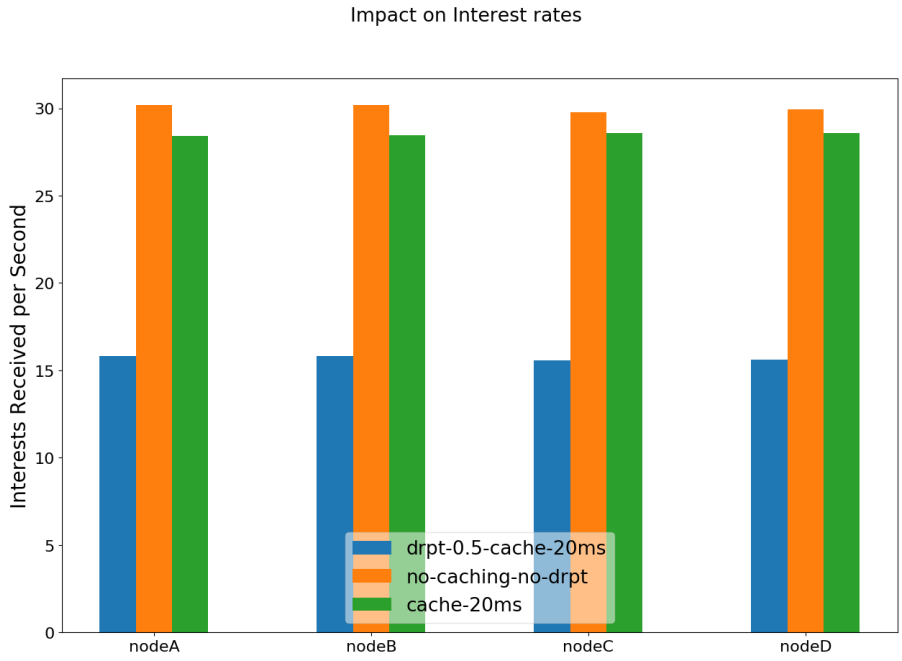

Figure E.5: Effects of enabling caching and DR publisher throttling on the Interest rate seen by nodes in the *square* topology.