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

School of Computer Science and Statistics

An Investigation into Building a Multiplayer Online Game Using Named Data Networking

Stefano Lupo
14334933

March 29, 2019

An MAI Thesis submitted in partial fulfillment
of the requirements for the degree of
MAI Computer Engineering

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Acknowledgements

Thanks to Dr Webber, Junxiao Shi

Contents

1 Abstract **11**

2 Introduction **12**
2.1 Background 12
2.2 Project Scope 12
2.3 Research Questions 12
2.4 Dissertation Outline 12

3 State of the Art **13**
3.1 Named Data Networking (NDN) 13
3.1.1 NDN Primitives 13
3.1.2 NDN Packet Structures 14
3.1.3 NDN Basic Operation 16
3.1.4 Names 18
3.1.5 Routing and Forwarding 19
3.1.6 In-Network Storage 20
3.1.7 Forwarding Strategies 21
3.1.8 Security 21
3.1.9 NFD 22
3.1.10 NLSR 24
3.1.11 NDN Tools and Libraries 26
3.1.12 NDN Benefits in a MOG Context 26
3.1.13 Host Based Applications using NDN 29
3.1.14 Real-Time Applications using NDN 29
3.1.15 Dataset Synchronization (DS) in NDN 30
3.2 Mutliplayer Online Games (MOGs) 35
3.2.1 Game Development 35
3.2.2 MOG Data Taxonomy 35
3.2.3 MOG Architectures 38
3.2.4 Dead Reckoning 40

3.2.5 Interest Management . . . 41
3.3 Closely Related Projects . . . 43

4 Problem Statement **47**
4.1 Primary Objectives . . . 47

5 Design **48**
5.1 CoolGame - a 2D, top down, shooting game . . . 48
5.1.1 Design Requirements of CoolGame . . . 49
5.2 CoolGame Data Taxonomy . . . 50
5.3 Player Discovery . . . 52
5.3.1 Benefits . . . 53
5.4 CoolGame Sync Protocol . . . 54
5.4.1 Motivation . . . 54
5.4.2 Name Schema . . . 55
5.4.3 Game Object Sync Protocol in Operation . . . 57
5.4.4 Benefits . . . 59
5.5 Interaction . . . 59
5.5.1 Name Schema . . . 59
5.6 Dead Reckoning . . . 59
5.7 Interest Management . . . 59

6 Implementation **60**
6.1 Frontend . . . 60
6.1.1 LibGDX . . . 60
6.1.2 Ashley - Entity Management System) . . . 60
6.1.3 Guice . . . 60
6.1.4 Reconcilers . . . 60
6.1.5 Creators . . . 60
6.2 Backend . . . 60
6.2.1 NDN Configuration . . . 60
6.2.2 Sequence Numbered Cache . . . 60
6.2.3 Concurrency . . . 60
6.2.4 Protos . . . 60
6.2.5 Linkage between game and backend . . . 60
6.2.6 Profiling . . . 61
6.2.7 Metrics . . . 61
6.3 Testing Implementation . . . 61
6.3.1 Automation Script . . . 61
6.3.2 Docker . . . 61

6.3.3 NLSR . . . 61
6.3.4 Latency Calculations . . . 61
6.3.5 Analytics . . . 61
6.3.6 AWS . . . 61

7 Evaluation **62**
7.0.1 Round Trip Times . . . 62
7.0.2 Effects of Enabling Caching . . . 62
7.0.3 Effects of Interest Aggregation . . . 63
7.0.4 Effects of Forwarding Strategy . . . 63
7.1 Overhead . . . 63

8 Conclusion **64**
8.0.1 Further Work . . . 64

Appendices **70**

A Invertible Bloom Filters **71**

List of Figures

3.1 NDN Packet Structure [5] 14
3.2 NDN operation on receiving Interest 17
3.3 NDN operation on receiving Data 18
3.4 An example NFD setup 23
3.5 NLSR LSA structure [22] (adapted) 25
3.6 Interest aggregation, native multicast and in-network caching 27
3.7 ChronoSync digest tree for a dataset synced across Alice, Bob and Ted [37] 32
3.8 Taxonomy of MOG Data 36
3.9 Tile interest map in a 3D game world 43
3.10 Core areas associated with the research project 43
3.11 Matryoshka broadcast discovery namespace 45

5.1 CoolGame - a 2D, top down game developed to facilitate research into MOGs using NDN 49
5.2 Taxonomy of MOG data with corresponding data in CoolGame 51
5.3 Name schema of CoolGame's game object sync protocol 55

List of Tables

List of Code Listings

Nomenclature

NDN	Named Data Networking
CCN	Content Centric Networking
LSR	Link State Routing
NLSR	NDN Link State Routing
NPC	Non Playable Characters
DS	Dataset Synchronization
DSP	Dataset Synchronization Protocol
NLSR	NDN Link State Routing
NLSR	NDN Link State Routing
NLSR	NDN Link State Routing

TODOs

- Quoute verbatim references 13
- ref sync protocol 19
- Reference Player Discovery multicast 21
- This section isn't great 24
- Change these to a,b,c,d 27
- custom sync protocol ref 28
- IBF appendix 34
- Reference games 35
- These are weak 54

1 Abstract

2 Introduction

[1]

2.1 Background

Existing IP based internet: host abstraction, where it comes from, How it has scaled, What it supports / doesn't support (mutlicast etc), History of ICN (CCN –> NDN, Parc etc), Thin waist

2.2 Project Scope

talk about limitations etc

2.3 Research Questions

2.4 Dissertation Outline

3 State of the Art

blah blah blah.. overview of all sections

3.1 Named Data Networking (NDN)

Quoute verbatim references

Today, most networks 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 [2]. 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 project is a continuation of an earlier project known as Content-Centric Networking (CCN) [3]. 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 alternative to IP, maintaining many of they key features which made it 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.

3.1.1 NDN Primitives

In NDN, as the name suggests, every piece of data 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 under this name, the camera in the kitchen for example, will receive this request and can respond by sending the data encapsulated in a Data packet, with the name field set to the name of the interest.

NDN communication is entirely driven by consumers who request data by sending interests and any unexpected data packets which reach NDN nodes are simply ignored.

3.1.2 NDN Packet Structures

As outlined in the NDN Packet Specification [4], 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 3.1, where red fields represent required fields and blue fields represent optional fields.

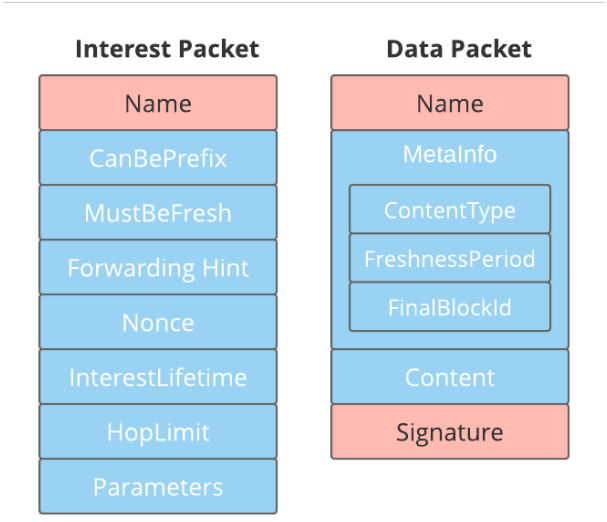

Figure 3.1: NDN Packet Structure [5]

The fields contained in NDN Interest packets are outlined below.

Name The name of the content the packet 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 CS entry whose *FreshnessPeriod* has expired.

ForwardingHint This defines where the packet should be forwarded towards if there is no corresponding FIB entry. Due to limited capacity of the FIB, only a small number of name prefixes can be stored and the *ForwardingHint* field can aid in mapping application data name prefixes to sets of globally "reachable" names [6]. This field typically represents an ISP prefix and is used to tackle the routing scalability issues present in NDN [7].

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 [4].

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 data to parameterize an Interest packet.

The fields contained in NDN Data packets are outlined below.

Name The name of the content the 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.

3.1.3 NDN 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 3.1.5), enabling a single Interest to be sent upstream through multiple interfaces.

As discussed in section 3.1.5, NDN uses *stateful forwarding* and the PIT is the data structure which maintains the forwarding state. This table 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.

Finally, the CS is used to cache data 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 offers 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 running on a machine producing data under a certain namespace.

The operation of an NDN node on receipt of an Interest packet is shown in figure 3.2.

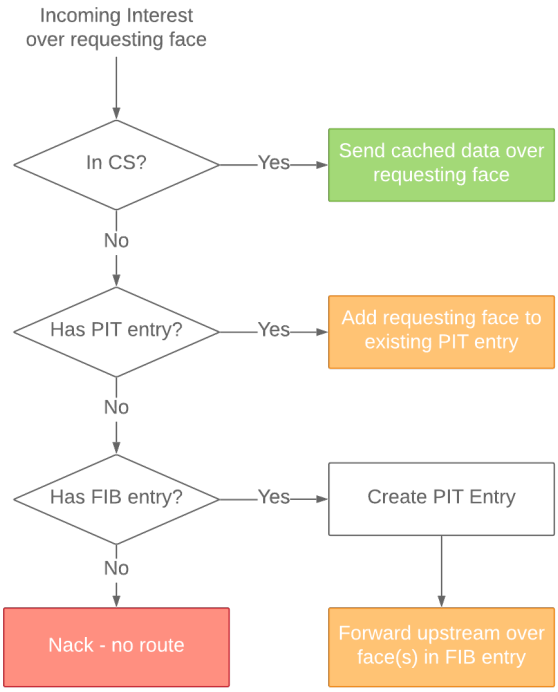

Figure 3.2: NDN operation on receiving Interest

On receiving 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 requested and forwarded upstream. Thus, 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.

The operation of an NDN node on receipt of a Data packet is shown in figure 3.3.

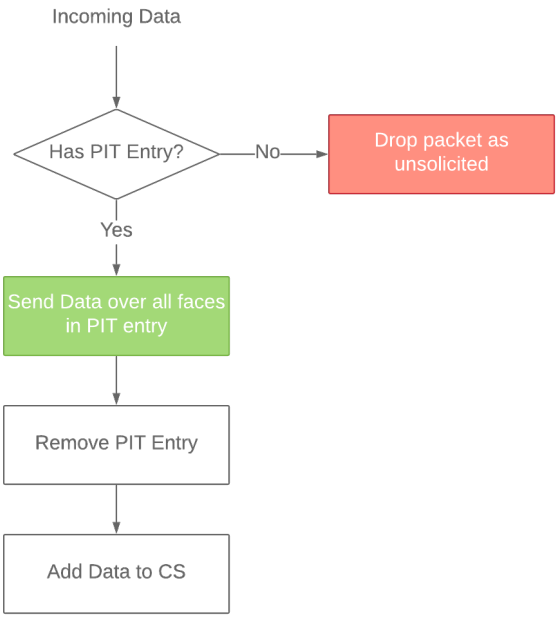

Figure 3.3: NDN operation on receiving Data

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 an Interest for this piece of Data. This means the Data 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.

3.1.4 Names

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 mechanism or agreed upon convention between a data producer and consumer to allow consumers to fetch data [8].

The names of Data packets can be more specific than the names of the Interest packets which trigger 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

ref sync protocol

.

3.1.5 Routing and Forwarding

Longest prefix, hierarchical naming (ndn project has some stuff on p3 2.2.1 names), NLSR, stateful forwarding can allow routers to measure performance of routes and update things accordingly (they see packets going out AND COMING BACK unlike Implementations, also it is what enables multicast and in network caching)

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), but 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 [9] in order to accomplish routing packets by name and not address, as seen by the usage of a PIT.

As discussed in section 3.1.4, 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 [9].

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 [10]. 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)* [11] 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, Interest packets cannot loop. As discussed in section 3.1.2, 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 3.1.10)

Native Multicast

As discussed in section 3.1.3, 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.

3.1.6 In-Network Storage

As discussed in section 3.1.3, 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 [5]. The CS of routers provides a mechanism for opurtunistic in-network caching, which can help reduce traffic load for popular content.

NDN also supports larger volume, persistent in-network storage in the form of *repo-ng* [12], which supports typical remote dataset operations such as reading, inserting into and deleting data objects [13]. This mechanism provides native network level support for Content Delivery Networks (CDN) [5] and can allow applications to go offline for longer periods of time while their content is served from in-network repositories.

3.1.7 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*, *NCC*, *Multicast* and *Client Control* [14]. 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. For 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. For 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 application's correctness can also be affected by the forwarding strategy [15]. 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* should be used in this context. This is outlined further in .

Reference Player Discovery multicast

3.1.8 Security

The aspect of security in NDN is the shift from attempting to secure communication channels to focusing on securing the data itself at production time.

Typically, the main form of secure communication in the Internet today is using the Transport Layer Security (TLS) protocol [16] along with the Transmission Control Protocol (TCP) over IP (TCP/IP). As discussed in section 3.1, TCP/IP is a mechanism for allowing communication between two nodes in a network. TCP/IP sets up a communication channel between the hosts and TLS is used to secure that channel.

NDN on the other hand focuses on securing the Data packets produced in response to Interests. As shown in section 3.1.2, NDN Data packets must contain a *Signature* field. A cryptographic signature is generated using the producers public key, binding the producer's name to the content. [17].

As NDN uses public key cryptography, all applications and nodes must thus have their own set of keys and a means 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 its key and certifies the ownership of this key [17]. Trust Anchors are the certificate authority for a given NDN namespace. NDN nodes can then 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.

3.1.9 NFD

In order to provide the NDN functionality, the *NDN Forwarding Daemon (NFD)* was developed. NFD is a network forwarder that implements the NDN protocol [18]. The NFD thus implements all of the features described in section 3.1.3 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, 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/TCP tunnels or Unix sockets. This allows applications using the NDN Common Client Libraries (see section 3.1.11) 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 applications using the NFD is shown in figure 3.4. 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 3.1.10). Finally, node A can specify the *Forwarding Strategy*, or use the default of *Best Route*.

As nodeB 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.

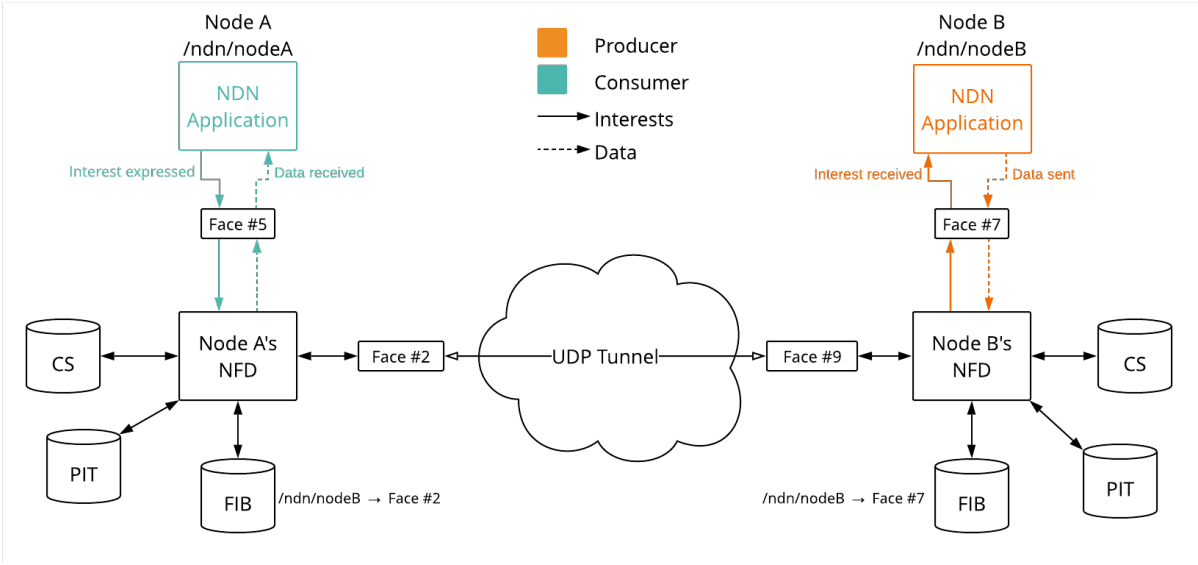

Figure 3.4: An example NFD setup

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 in the process.
5. Node B's NFD then finds the FIB entry created for */ndn/nodeB* when nodeB 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.

3.1.10 NLSR

This section isn't great

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 [19].

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 [20][21]. 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 and the edge weight (link cost) for each of those neighbour connections. Unlike the *HELLO* messages for neighbour discovery, routers will forward LSPs from a specific router to their direct neighbours, **once per sequence number**. Thus, routers need to maintain state containing the most recent LSP it has seen for each router in order to determine whether or not a given LSP is newer than what it has already seen and thus whether or not to forward this version. 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 3.1.3). Thus it uses Interest and Data packets as the only form of communication between routers. NLSR differs from the traditional IP based LSR protocol in the following ways as it uses hierarchical naming schemes for routers, keys and updates, it uses a hierarchical trust model, it uses ChronoSync to disseminate routing updates (see section 3.1.15) and it supports multipath routing [22].

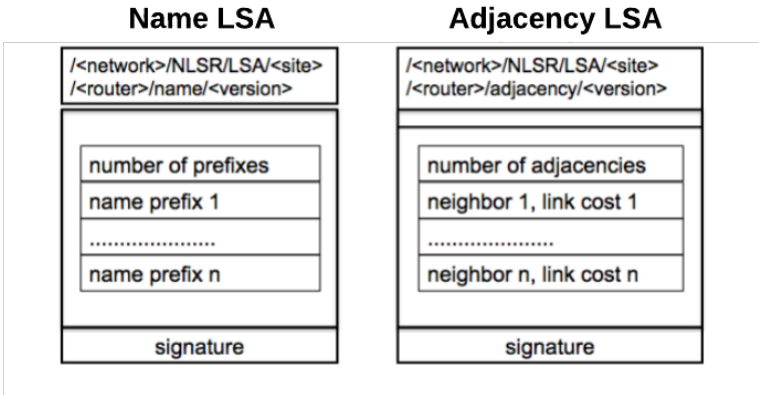

Figure 3.5: NLSR LSA structure [22] (adapted)

As seen in figure 3.5, 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, while *adjacency* LSAs contain the list of neighbours a router has as well as their associated link costs. LSA dissemination is essentially a dataset synchronization problem and thus NLSR uses NDN's ChronoSync protocol (see section 3.1.15) to synchronize the LSAs. *Name* LSAs can be updated as registered prefixes change, while *adjacency* LSAs can change as routers go offline and come back online. In the steady state, all routers will maintain an outstanding sync Interest, containing the same digest of the LSA dataset. This outstanding sync Interest will be named */<network>/nlsr/sync/<digest>* and the forwarding strategy of */localhop/com/nlsr/sync/* is set to multicast on all routers, allowing all routers to receive sync updates. 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 if the LSA. Other routers can then fetch this updated LSA using a standard Interest packet when convenient.

As with LSR, NLSR is responsible for building the FIB and thus requires a mechanism to discover adjacent routers. This is accomplished by setting the forwarding strategy of *<network>/nlsr/LSA* to multicast. When a new router receives the first response to the sync Interest it expresses, it can request the Data using the corresponding name and this Interest will be multicasted to all of the router's adjacent neighbours. This is required as the router's FIB may not have an entry for the corresponding name. However, this broadcast should not have 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.

3.1.11 NDN Tools and Libraries

In order to facilitate the development of NDN applications, the API specified in the NDN Common Client Libraries Documentation [23] has been ported to a variety of popular languages such as C++, Python and Java. Several command line tools under the NDN Tools project [24] 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 [25] has also been developed to facilitate experimentation using the NDN architecture. This project has been under continuous development since 2012 and has been used by hundreds of researchers around the world [26].

Another project built by the NDN team is Mini-NDN [27]. Mini-NDN is based on the popular network emulation tool Mininet [28] 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.

3.1.12 NDN Benefits in a MOG Context

Interest aggregation, in network caching, native multicast, multipath forwarding.

As outlined in section 3.2, MOGs can be built using a variety of architectures, can have a variety of different data types and require extremely performant 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*.

Interest Aggregation

As discussed in section 3.1.3, 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. Thus, as the Interests from separate consumers are forwarded closer and closer to the producer, it is 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 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.

This also provides a benefit from the point of view of game players as publishers, as they should only see and need to respond to **one instance** of each Interest, provided consumers only request Data within the freshness period, as Interests will be aggregated at their local NFDs as well. An example of Interest aggregation is seen at $time = t2$ of figure 3.6.

Change these to a,b,c,d

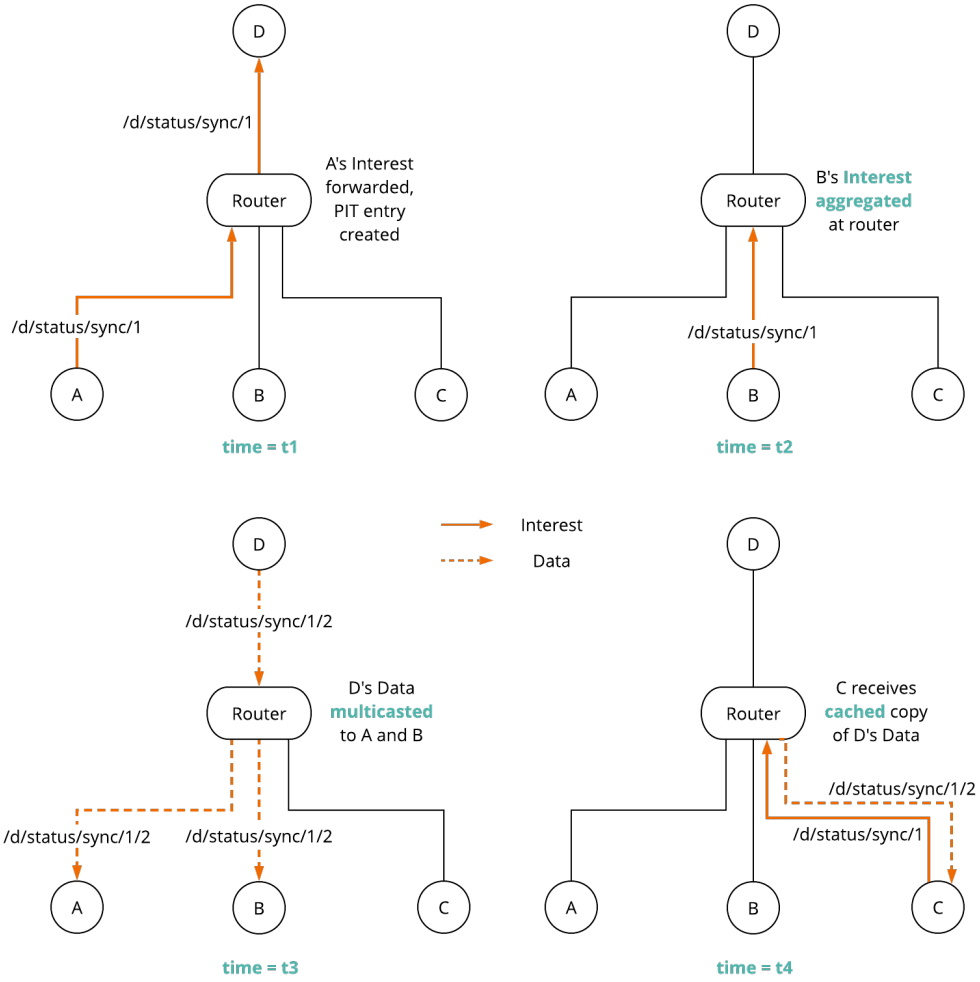

Figure 3.6: Interest aggregation, native multicast and in-network caching

Native Multicast

The core concept behind multicast is producing a piece of data once and having it reach multiple consumers and NDN provides this natively. Native multicast is a direct result of NDN's stateful forwarding plane, Interest aggregation mechanism and in-network caching. 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. An example of native multicast is seen at $time = t3$ of figure 3.6.

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, reducing the round-trip-times of fetching updates from the network and thus the overall latency of the MOG. Although static content (see section 3.2.2) 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.

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 , 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, if a consumer falls slightly behind 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 producer. This would likely occur concurrently to when other consumers are requesting the **next** Data packet, meaning the consumer will continue to remain behind and continue to essentially **double** the number of Interests seen by the producer and largely increase the amount of network traffic required for a certain sequenced piece of Data.

custom sync protocol ref

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 hope-

fully catch up with the other consumers, or continue to obtain cached copies previously fetched. An example of in-network caching is shown in figure 3.6 at $time = t4$.

3.1.13 Host Based Applications using 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., 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 [29]. 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) [30]. 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.

Another difficulty associated with conventional IP is managing mappings from IP addresses to actual users. VoIP requires mappings from user identities to endpoint IP address at multiple points in the network [29]. However, in the content centric approach, a user's identity is fully defined by the key used to sign data that it creates,

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 study is required.

3.1.14 Real-Time Applications using 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 architecture struggles to facilitate applications in which producers much stream real time data to several consumers, requiring an end-to-end connection between the producer and each consumer. These applications are also becoming more and more common with the advent of streaming platforms, such as those provided by television providers.

As discussed by Gusev et al [31], 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 real-time applications using NDN have been developed. NDN-RTC is [32], a real-time video conferencing library for NDN built on top of WebRTC. Voice over Content Centric networking (VoCCN) [29] as discussed in section 3.1.13, is an NDN equivalent to VoIP. Real-time Data Retrieval (RDR) [33] outlines a protocol for allowing consumers to obtain the most recent frame published by a producer and for pipelining Interests for future frames.

3.1.15 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 provided all participants with a common view of the messages sent to the chat room, the underlying dataset must be synchronized by a synchronization protocol. The importance of dataset synchronization protocols (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. One of the goals of this 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.

However 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 simply 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 [34]. Several protocols have been developed to achieve this including CCNx Sync 1.0 [35], iSync [36], ChronoSync [37], RoundSync [38] and PSync [39]. 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 3.7.

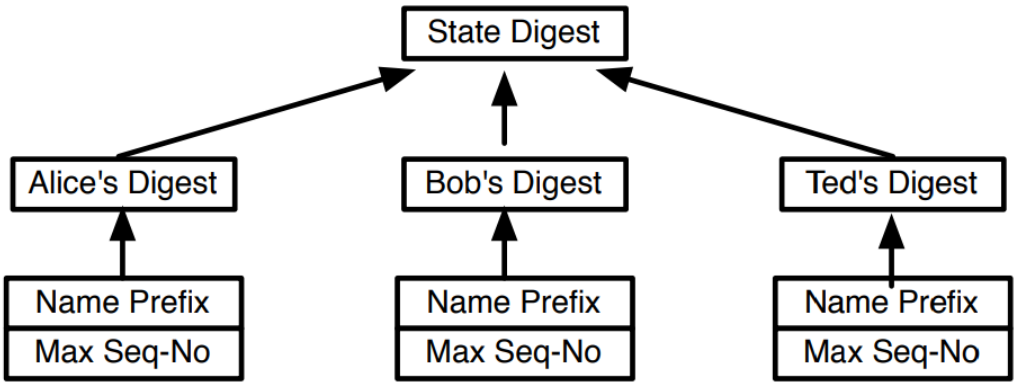

Figure 3.7: ChronoSync digest tree for a dataset synced across Alice, Bob and Ted [37]

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 *SyncInterest*. 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 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 Data 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 from their point of view and express a new SyncInterest, which will match the Interest 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 one outstanding Sync-

Interest 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 of them 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) it lets each 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 [38].

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 [38].

Although RoundSync appears to offer several benefits over ChronoSync, the only available implementation of the protocol is for use with ndnSIM [40].

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.

IBF appendix

PSync uses BFs to store the *subscription list* of subscribers. PSync uses 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.

Producers in PSync maintain **no state** regarding their consumers and instead store a single IBF for all consumers, providing scalability under large number of consumers [39]. 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 it's current producer state from the producer state contained in the SyncInterest (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.

3.2 Mutliplayer Online Games (MOGs)

3.2.1 Game Development

Reference games

A huge variety of video game engines and 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 was used in a number of immensely successful games such as Half-Life, Team Fortress 2, Portal 2 and Counter Strike: Source. The Unity game development platform was used to develop major titles such as Kerbal Space Program and Hearthstone: Heroes of Warcraft. EA DICE's Frostbite engine has been used in a variety of genres ranging from sports titles such as FIFA 19 to first person shooters such as Battlefield 4.

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 [41]. 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.

3.2.2 MOG 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.

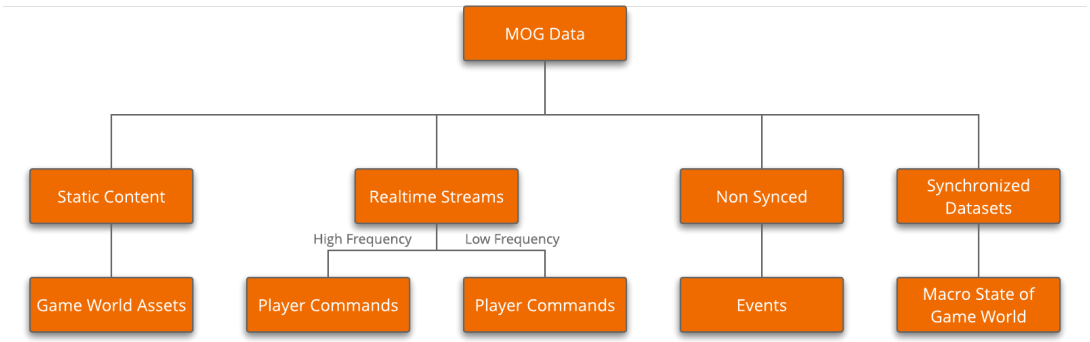

Figure 3.8: Taxonomy of MOG Data

The overall taxonomy of MOG data is shown in figure 3.8 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 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 a new player coming into view, the textures are likely to be cached by routers in the network, as other players may have previously required them. However, in comparison to the other categories of data, the frequency of fetching this data is so low that improvements in network performance regarding this data would likely have a negligible impact on the overall network performance.

Realtime Streams

The second category of data found in MOGs is real-time data which is sent repeatedly over time, at a somewhat consistent interval. Due to the likely 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 in player commands. However, these can be further subdivided by the frequency of data updates.

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 real-time data stream of data 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 remote players must be informed about, 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 associated synchronization aspect of this data over time - the player simply announces to the network that they are reloading their weapon by publishing an immutable, short lived event.

Syncrhonized 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 stricter consistency amongst game players and can therefore use more expensive protocols which would not be suitable for other data types.

3.2.3 MOG 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 connected players have a local set of game objects which are shown to the player, though the distribution of primary and secondary copies depends on the game's architecture [42]. All updates to game objects are performed on **primary copies only**. The results of these update operations are the sent to all players who require the latest copy of the game object, updating their secondary copies accordingly.

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 in comparison 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 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 they are still the most commonly used today [42]. 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 [43]. 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 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* [44]. 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 [45] and Scribe [46] 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 the lack of a single authority. This requires

much more complex protocols to synchronize the state of shared objects while presenting a responsive simulation of the game world [47]. As players are responsible for accepting, rejecting and performing updates on primary copies, P2P based architectures are much more vulnerable to cheating.

3.2.4 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 high frequency update requirement, coupled with the amount of remote game objects that need updating in modern MOGs mean 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 [47], "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, using the basic equations of linear motion, 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.

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 the future position, defined by the trajectory built using the position and velocity contained in the

remote update. An example DR convergence algorithm is one which builds a trajectory using the newly received update packet. A target point is then chosen on that trajectory a configurable number of time steps away. A smoothing function is then used to build another trajectory from the previously extrapolated position, to the new target point, allowing the object to smoothly reach the targeted future point [48].

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 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 [49].

3.2.5 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. Finally, the game world may 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* in nature, as described previously. However, a somewhat assumed form of IM in MOGs is *temporal* in nature, in that MOGs are essentially real-time 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. Finally, there are also opportunities to employ IM by exploiting certain features of the actual game. For example, in a shooting based 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 performing IM to determine 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 computation overhead [50]. 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 surplus 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 player for that game object can be slowed and the contents of the updates can be diminished to only containing 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 produces false positives for game objects which the player should not be interested in. Tile based IM solves this problem by taking the game world into account [51].

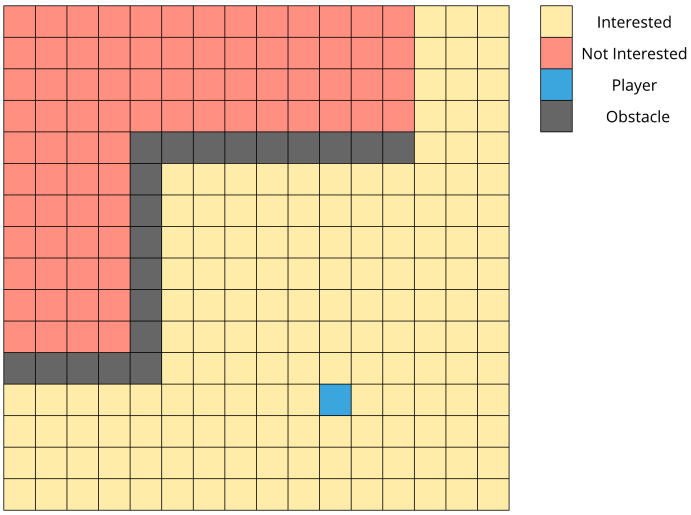

Figure 3.9: Tile interest map in a 3D game world

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 3.9 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).

3.3 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 3.10.

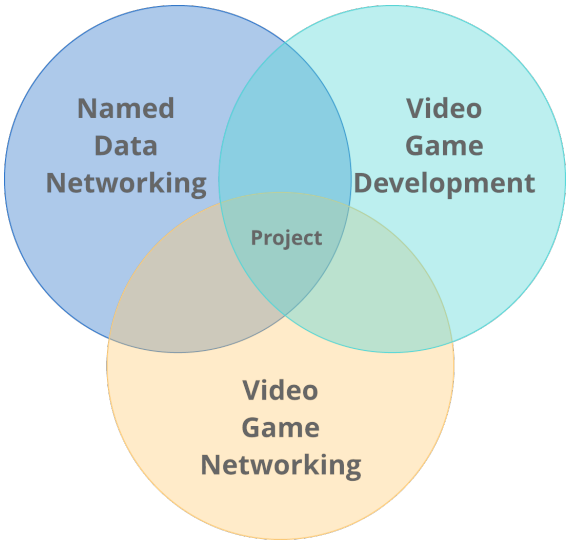

Figure 3.10: Core areas associated with the research project

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 [52]

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 [35] and the corresponding data repository as a means of 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 order of asset discovery is unnecessary.

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 is no publicly source code available.

Matryoshka [53]

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 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 3.11. Finally, players can periodically express Interests for the game objects in their partition, using the set of names they have discovered.

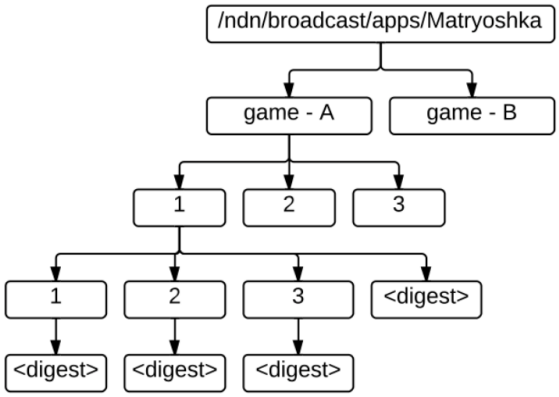

Figure 3.11: Matryoshka broadcast discovery namespace

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 [54]

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.

4 Problem Statement

What to do for project after SOTA?

4.1 Primary Objectives

5 Design

Outline this section + some filler

5.1 CoolGame - a 2D, top down, shooting game

To allow for the testing of NDN in a MOG context, a simple game was required. Although a variety of open source games of various complexities exist, building the front end of the game from scratch as opposed to adding a networking module to an existing game was chosen for the following reasons:

- The planned scope of the front end of the game was very small, meaning the time investment to build the front end of the game would not be substantial, in comparison to the rest of the project.
- 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.
- Designing and implementing the game from scratch would allow for a deeper understanding of the overall system.
- Although the networking aspects of the actual game are decoupled from the front end 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 front end when targeting an ICN based back end and these optimizations would never be explored if an off the shelf game was used,
- Depending on the available time, the game could grow in complexity to support other features which are interesting from a networking perspective.

5.1.1 Design Requirements of CoolGame

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, CoolGame 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 CoolGame 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 real-time.
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 remove players.
5. The player must be able to interact with remote players and the interaction must be visible to all remote players.

CoolGame was designed to meat each of the requirements defined above and a screenshot from the game is shown in figure 5.1.

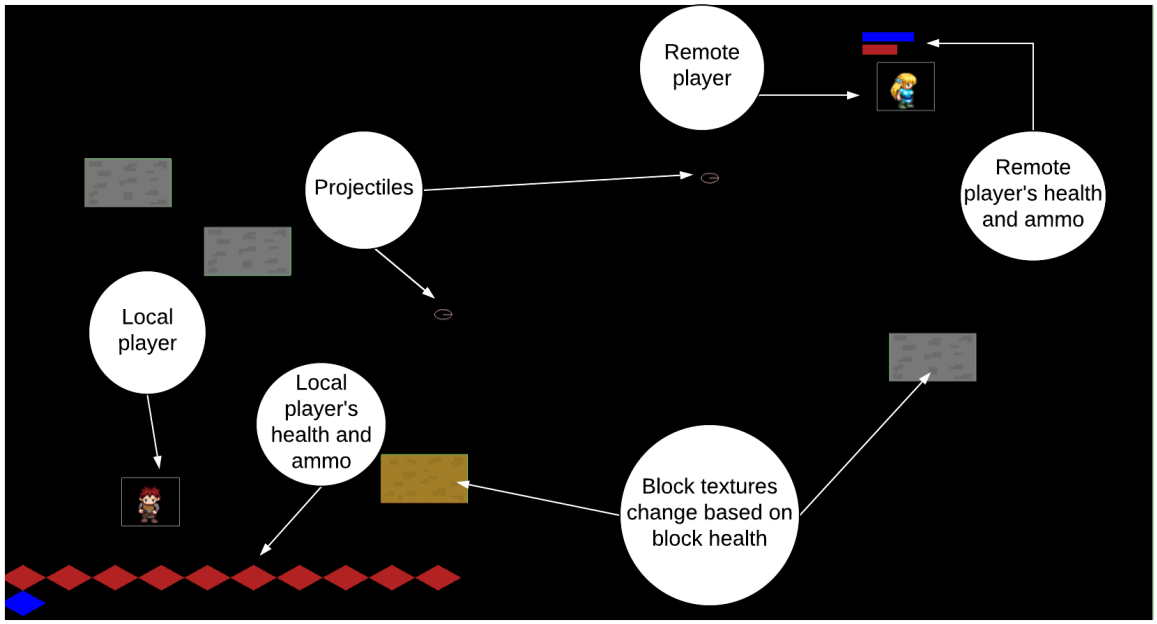

Figure 5.1: CoolGame - a 2D, top down game developed to facilitate research into MOGs using NDN

CoolGame 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 5.1. 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 5.1 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 5.1 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 5.1, 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.

5.2 CoolGame Data Taxonomy

The taxonomy for MOG data is outlined in section 3.2.2. 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 CoolGame is shown in figure 5.2.

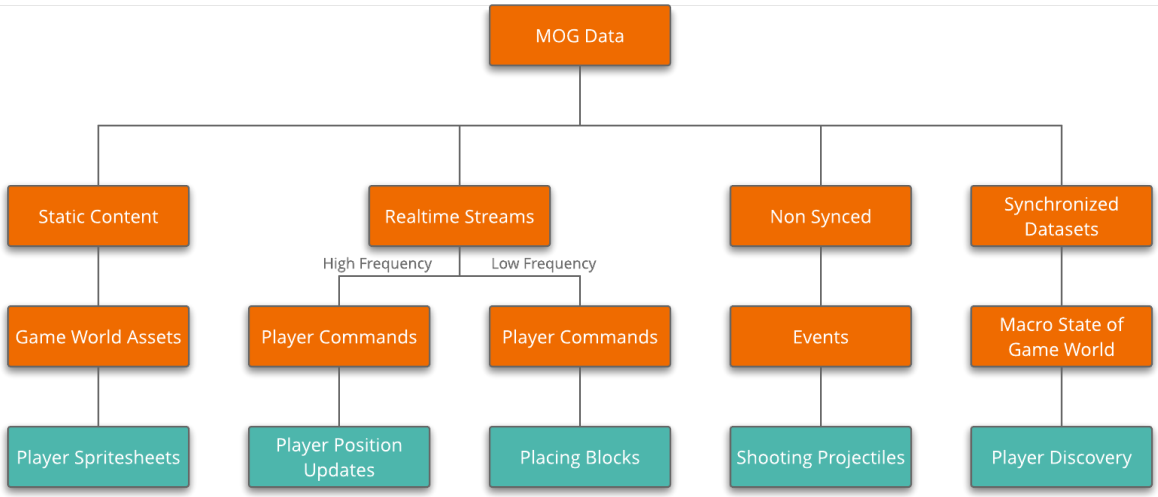

Figure 5.2: Taxonomy of MOG data with corresponding data in CoolGame

As shown in figure 5.2, each of the data types produced by CoolGame 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 5.2, real-time 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, those 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 CoolGame 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 interaction with the game world boundary, they are automatically destroyed locally by all players. However, on interaction with a remote player or block created by a remote player, the projectile is destroyed and subsequent action is taken through the Interaction API (see section 5.5). 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.

Syncrhonized 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.

5.3 Player Discovery

As shown in the taxonomy of CoolGame's data (figure 5.2), the problem of player discovery is one of dataset synchronization (DS). As discussed in section 3.1.15, 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 CoolGame. 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 [23], 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 3.1.15), 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 CoolGame - the *gameId* and the *playerName*. The *gameId* is chosen ahead of time and allows the player to choose the instance of CoolGame 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 *playerName* of all connected players.

As outlined in section 3.1.15, 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 CoolGame is */<game_prefix>/discovery/broadcast*. As discussed in section 3.1.7, 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 CoolGame, the player discovery data is named */<game_prefix>/discovery/<player_name>*. Currently, 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.

5.3.1 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 the characteristics of the player discovery data allow for the limitations inherent in the ChronoSync protocol. The current player discovery mechanism is naive in that it is performed globally across all players in a given game instance.

Matryoshka (see section 3.3) 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 3.2.2), player discovery is an extremely light weight task. Currently, only data required by CoolGame 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 CoolGame 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 be likely be required.

5.4 CoolGame 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, low latency synchronization of game objects over NDN.

5.4.1 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 and participants 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 paramount. 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:

These are weak

1. Players are only interested in the **newest instance** of a piece of named data. The real-time 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 their newest 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.

5.4.2 Name Schema

One of the most important aspects of designing an application or protocol which uses NDN is *naming*. 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 CoolGame's game object sync protocol is shown in figure 5.3.

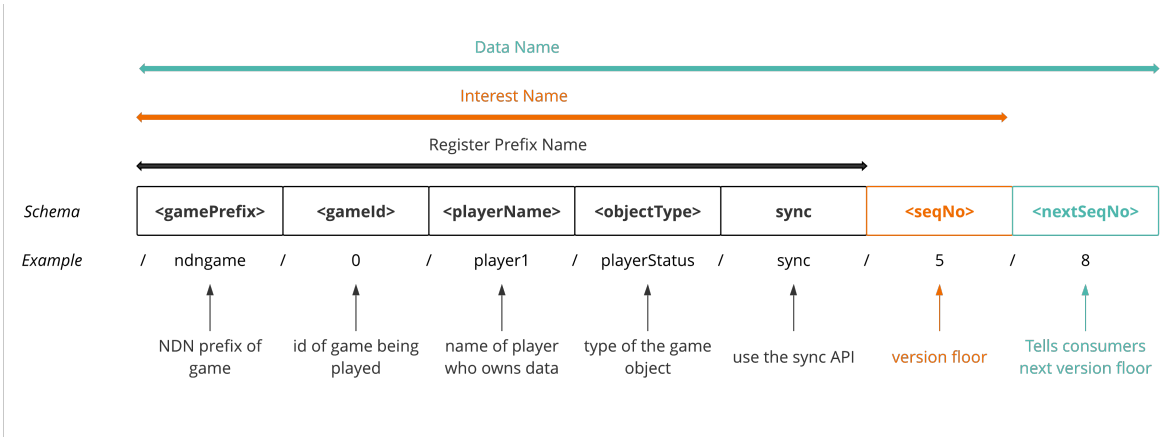

Figure 5.3: Name schema of CoolGame's game object sync protocol

As seen in figure 5.3, 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 CoolGame in the global NDN namespace.

gameId

This is used to allow for multiple instances of CoolGame 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 CoolGame.

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 5.3).

objectType

This specifies the type of the game object in question. In the current implementation of CoolGame, 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 5.5).

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.

5.4.3 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 the 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 it'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*

5.4.4 Benefits

The main benefit of the synchronization protocol is that it does not require a separate Interest / Data exchange for update notifications and update fetching.

5.5 Interaction

This is not handled in Egal Car

5.5.1 Name Schema

Things can't be parametrized in NDN without forming a gross name schema - e.g. interaction API can't send parameters. Potential solution is to have publisher's maintain outstanding interests towards consumers for interactions?

5.6 Dead Reckoning

General how I plan to do DR section 3.2.4 Dead reckoning impacted by caching? E.g. getting someone else's dead reckoned packet? Stefan mentioned this I cant decide if it matters

5.7 Interest Management

General how I plan to do IM section 3.2.5 Game world larger than viewpor

6 Implementation

Front end, backend developed in parallel. Local testing using single NFD, actual testing using docker, and aws

6.1 Frontend

6.1.1 LibGDX

6.1.2 Ashley - Entity Management System)

6.1.3 Guice

6.1.4 Reconcilers

6.1.5 Creators

6.2 Backend

6.2.1 NDN Configuration

CanBePrefix of Sync protocol Interest packets, broadcast namespace for chronosync. Interest life times, freshness periods.

6.2.2 Sequence Numbered Cache

6.2.3 Concurrency

6.2.4 Protos

Used to transport EMS entities and fed into reconcilers

6.2.5 Linkage between game and backend

diagrams of pubs / subs / game engine

6.2.6 Profiling

6.2.7 Metrics

Interest Management

Linear dropoff distance thing

Dead Reckoning

Cache kept on producers, system that runs it

6.3 Testing Implementation

6.3.1 Automation Script

Must be repeatable, used fixed seed for RNG

6.3.2 Docker

6.3.3 NLSR

Building topologies Automating players, simulators, INCREDIBLES

6.3.4 Latency Calculations

6.3.5 Analytics

Python script, use dropwizard metrics, uses NFd logs

6.3.6 AWS

7 Evaluation

Key things to examine: scale, overhead (packet size vs app data), latency Evaluation Matrix Push to breaking point

In order to examine the performance of the game, ...

Need to decide what game objects to use

7.0.1 Round Trip Times

As a bench mark, no caching, no DR, no IZF RTT for each topology

7.0.2 Effects of Enabling Caching

Discuss difficulties of maintaining fresh cache in MOG scenario where data changes are not predictable (freshness period etc), tree topology makes G should never really be getting any cache hits cause any data that is cached at G should have been forwarded to one of the intermediate routers who would then cache the data. For Dumbbell the intermediate routers should only be caching data on the OPPOSITE SIDE? If an interest arrives at F for C/D, it will be cached at F on the way back. However once it reaches E it will be cached there too. If A or B then request the interest it should be cached and served from E not from F. As in-network caching is one of the main benefits of NDN for typical use cases such a serving content, the impacts of using caching for the MOG were studied. Theoretically, enabling caching would directly impact the following

isher Interest Rate

Round Trip Times

ote Update Deltas

IT KIND OF MAKES SENSE THAT CACHE RATES ARE LOW AS ITS ALMOST ALL INTEREST AG. BUT ENABLING IZF AND DR COULD CUASE CACHE HITS ROUND TRIP TIMES WITHOUT DEAD RECKONING ARE ALMOST ENTIRELY DEPENDENT ON LOCALPLAYERSTAUTS PUBLISH RATE!!!!

7.0.3 Effects of Interest Aggregation

This is really not working the way I thought it would be :/ Compare interests received for each node's status to sum of interests expressed towards that node by all other nodes

7.0.4 Effects of Forwarding Strategy

Multicast shouldn't make a difference in tree like topology if interests are aggregated as there is only one upstream node from each route to a data source as defined by NLSR. On square however, producers should see all of the interests provided their local NFD's don't aggregate the interest which they dont seem to be.

7.1 Overhead

use ndndump to see packet sizes

8 Conclusion

8.0.1 Further Work

1. More rhobust interaction API, concurrent issues (e.g. two people shoot block at same time?)
2. Add support for NPCs - shouldn't be hard. Main interesting bit is distributing the set of NPCs across game players. Other than that its just another sync object

Things can't be parametrized in NDN without forming a gross name schema - e.g. interaction API can't send parameters. Potential solution is to have publisher's maintain outstanding interests towards consumers for interactions?

Bibliography

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

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

[3] 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.

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

[5] 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.

[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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

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

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

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

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

[22] 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.

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

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

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

[26] 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.

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

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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

[38] 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.

[39] 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.

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

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

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

[51] 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.

[52] 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`.

[53] 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`.

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

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.