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 27, 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

Contents

List of Figures

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
NLSR	NDN Link State Routing
NLSR	NDN Link State Routing
NLSR	NDN Link State Routing
NLSR	NDN Link State Routing
NLSR	NDN Link State Routing

TODOs

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

3 State of the Art (15 to 25)

3.1 Named Data Networking

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.

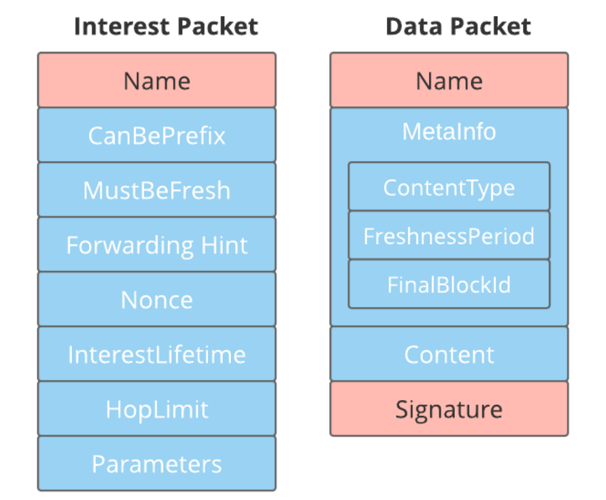

Figure 3.1: NDN Packet Structure [5]

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.

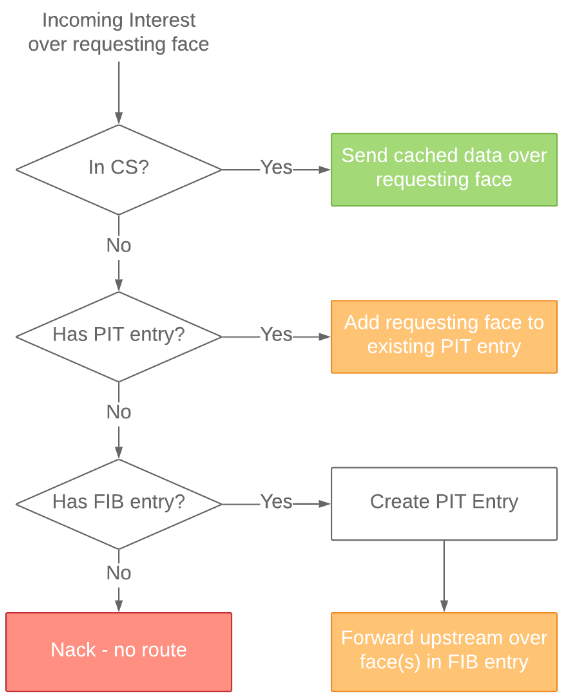
Incoming Interest over requesting face
In CS?
Yes
Send cached data over requesting face
No
Has PIT entry?
Yes
Add requesting face to existing PIT entry
No
Has FIB entry?
Yes
Create PIT Entry
No
Nack - no route
Forward upstream over face(s) in FIB entry

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.

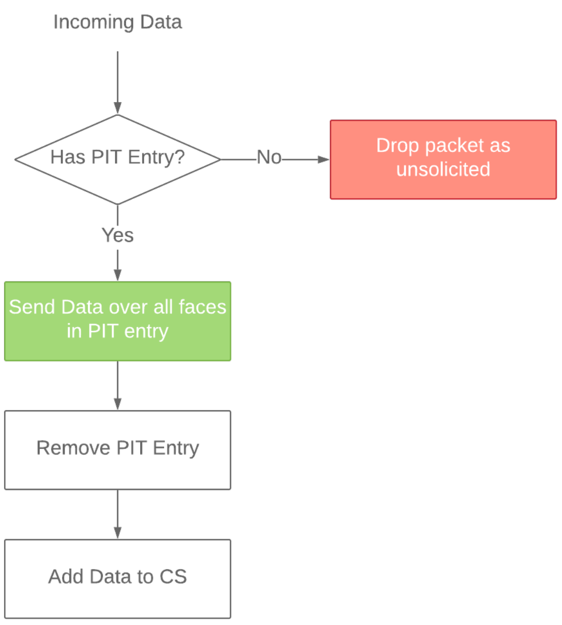
Incoming Data
Has PIT Entry?
No
Drop packet as unsolicited
Yes
Send Data over all faces in PIT entry
Remove PIT Entry
Add Data to CS

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 $/ndn/test/<sequence\text{-}number>$. 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.

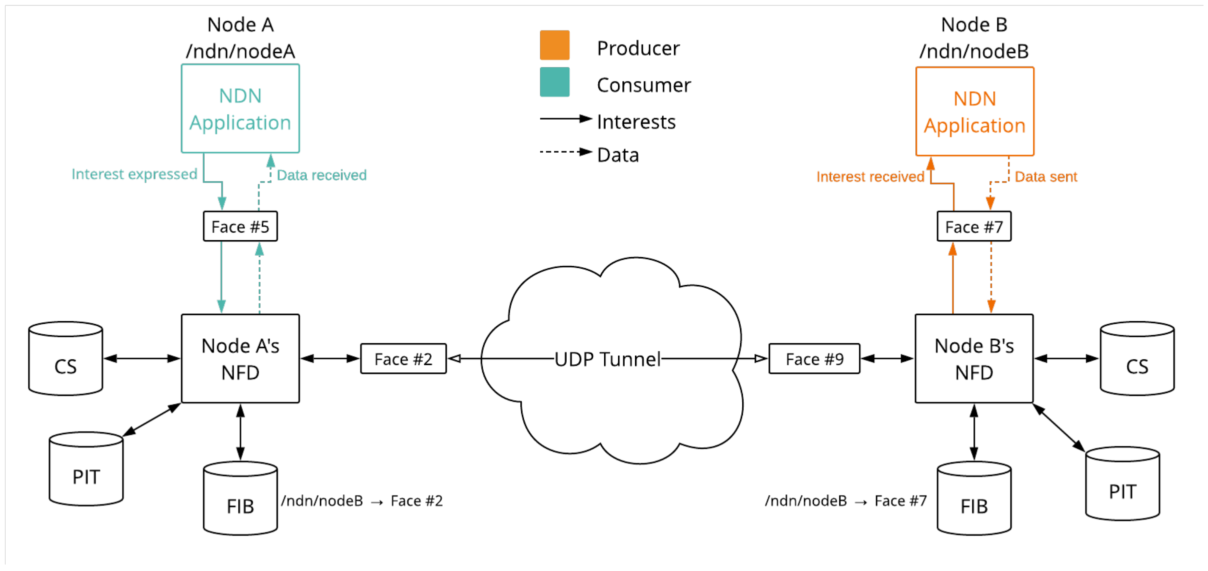

Figure 3.4: An example NFD setup

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].

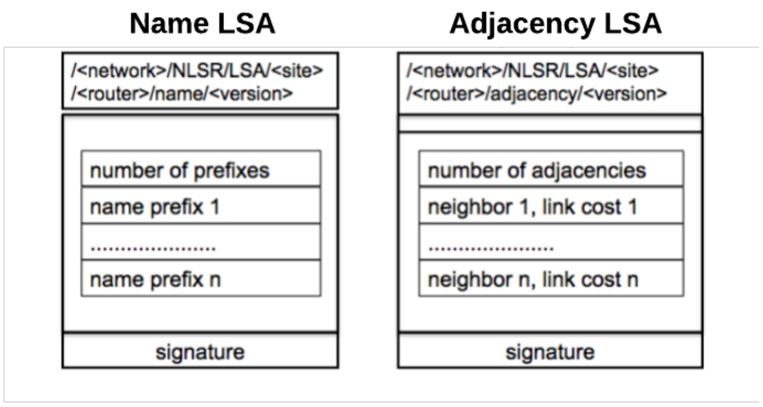

Figure 3.5: NLSR LSA structure [22] (adapted)

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.

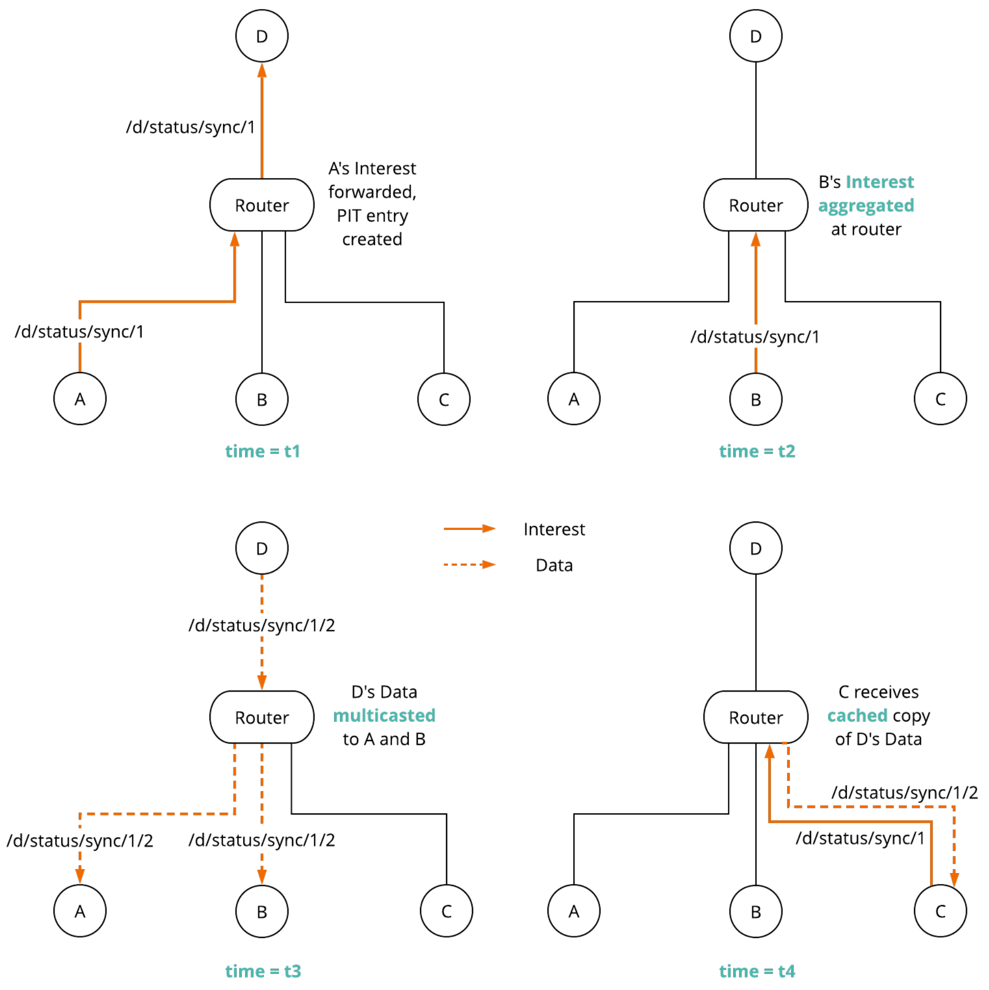

Figure 3.6: Interest aggregation, native multicast and in-network caching

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 3.2.3) 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 pro-tocol 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 hopefully 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 Distributed Dataset Syncrhonization 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 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 dataset synchronization in NDN has been conducted. One of the goals of this research is to abstract away the need for NDN application developers to consider dataset synchronization.

Traditionally, IP based solutions for dataset synchronization 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 dataset synchronization 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 dataset synchronization 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.

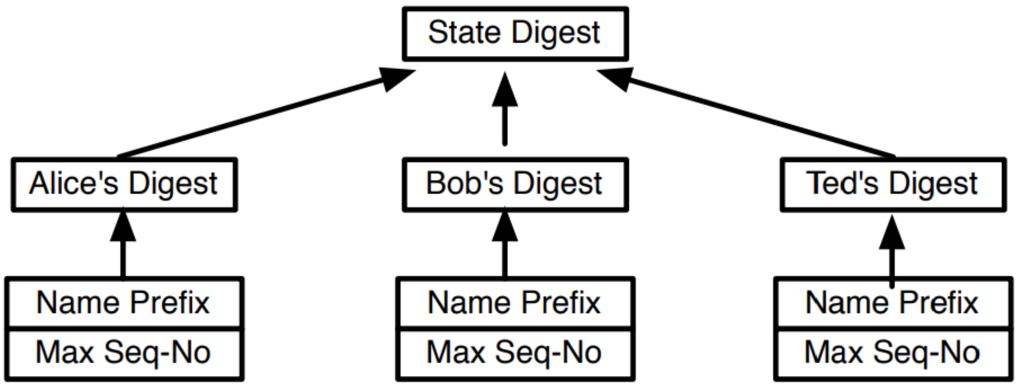

Figure 3.7: ChronoSync digest tree for a dataset synced across Alice, Bob and Ted [37]

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 SyncInterest on each link in the network. As a single Interest can only return a single Data packet in NDN, if two nodes produce two different SyncReplies for the same SyncInterest, only one 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

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 immensly 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 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 game 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 Entity Management Systems

3.2.3 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.

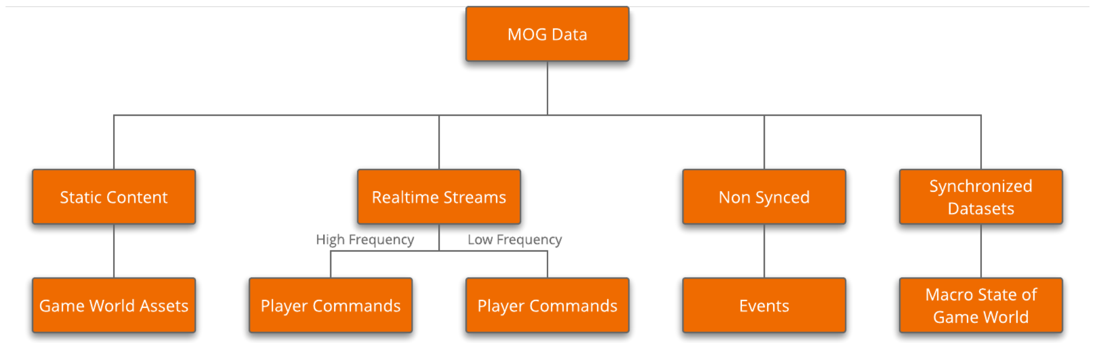

Figure 3.8: Taxonomy of MOG Data

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.

3.2.4 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 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 [46]. As players are responsible for accepting, rejecting and performing updates on primary copies, P2P based architectures are much more vulnerable to cheating.

3.2.5 Dead Reckoning

Reconciliation between dead reckoned position and remote update methods. Dead reckoning (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. [46]

3.2.6 Area of Interest Management

PubSub paper has interesting stuff on IZF Another mechanism for determining the entities of interest based on the players view of the game world is Binary Space Partition (BSP), as used in Quak

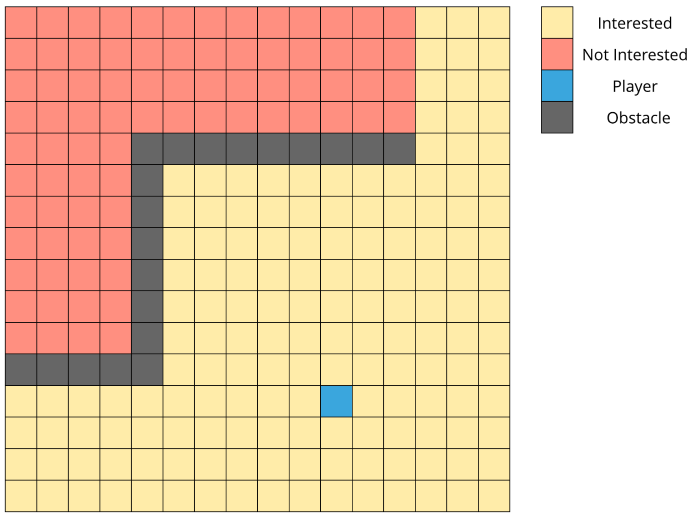

Figure 3.9: Tile-based area of interest management

3.2.7 Lag Compensation

Read up on lag compensation etc, Source Engine notes

3.3 Closely Related Projects

Egal Car [47]

Matryoshka [48]

NDNGame [49]

4 Problem Statement

5 Design

Explain the things MOGs must do

5.1 Player Discovery

5.2 Sync Protocol

Broadcast names (for discovery etc) need multicast forwarding strategy

5.2.1 Differences from existing protocols

outstanding interest catchup, only one writer, briefly discuss limitations of chronosync etc. Talk about how most ds sync protocols are designed for multiple writers such as Chronoysnc. Typically not the case in high frequency mog traffic. Psync closest to what I need, single producer, still needs two round trips

5.2.2 Naming Design

5.3 Interaction

This is not handled in Egal Car

5.4 Optimizations to Sync Protocol for Data types

5.4.1 PLayer Status

5.4.2 Projectile

5.5 Game Design

5.5.1 Protos

5.5.2 Very brief game functionality

5.5.3 Linkage between game and backend

diagrams of pubs / subs / game engine

5.5.4 Optimizations

Interest Zone Filtering

Dead Reckoning

Dead reckoning impacted by caching? E.g. getting someone else's dead reckoned packet? Stefan mentioned this I cant decide if it matters right now

5.5.5 Automation Script

Must be repeatable, used fixed seed for RNG

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?

6 Implementation

6.1 Building Game with LibGdx / Ashley

Remote update reconciliation

6.2 Java Backend

6.2.1 Sequence Numbered Cache

6.2.2 Concurrency

6.3 Testing

6.3.1 Docker

6.3.2 NLSR

Building topologies Automating players, simulators, INCREDIBLES

6.3.3 Latency Calculations

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

More rhobust interaction API, concurrent issues (e.g. two people shoot block at same time?)

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 pit-less 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] 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.

[47] 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`.

[48] 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`.

[49] 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.