University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

ICN with DHT Support in Mobile Networks

Timo Surbeck
Winterthur, Switzerland
Student ID: 15-701-733

Supervisor: Dr. Eryk Schiller, Sina Rafati
Date of Submission: September 15, 2021

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

In the course of the global digitalization in recent decades, the internet has become one of the most important and ubiquitous communication means. However, being based on an end-to-end connectivity principle, existing internet infrastructure is not optimal for data, *resp.*, content delivery use cases (such as Video-on-Demand): For such applications it is key to make the distribution of data from content producers towards multiple content consumers as efficient as possible. This fact has lead to the development of future internet architectures part of the ICN (Information Centric Networking) family: ICN architectures, such as NDN (Named Data Networking), can improve content delivery on a systemic level, by – contrary to the classical internet – focusing on identified data and deploying in-network caching techniques. Replacing the entire established internet with a novel architecture is however a non-trivial task, which is why this thesis considers a layered network architecture consisting of several smaller NDN-based mobile networks (*resp.*, domains): Thereby, independent domains are inter-connected using a Chord Peer-to-peer network running as an overlay on top of existing internet infrastructure. By using the NS-3 framework to develop a network simulation, which models real-world network characteristics, the performance of the proposed architecture is evaluated: This includes a comparison with a plainly NDN-based reference architecture, which reveals that the layered NDN & Chord approach is a valid and efficient alternative, if a global spanning NDN network cannot be realized, or, if NDN routing is subject to difficult conditions.

In den letzten Jahrzehnten wurde das Internet im Zuge der globalen Digitalisierung zu einem der wichtigsten und allgegenwärtigsten Kommunikationsmittel. Jedoch ist die bestehende Internet-Infrastruktur, welche auf einem auf Ende-zu-Ende Verbindungen basierendem Prinzip aufbaut, nicht optimal für Anwendungen, bei denen es um die Verteilung von Inhalten (*engl.* Content Delivery, z.B. Video-on-Demand) geht: Für solche Anwendungen gilt es, die Kommunikation zwischen Datenproduzenten und mehreren Datenkonsumenten effizienter zu gestalten. Dieser Umstand hat zur Entwicklung von zukünftigen Internetarchitekturen Teil der ICN (Information Centric Networking) Familie geführt: ICN Architekturen, z.B. NDN (Named Data Networking), steigern die Effizienz von Content Delivery Anwendungen indem sie – im Unterschied zum bestehenden Internet – systembedingt auf identifizierbare Daten fokussieren und Netzwerk-weit Caching-Techniken einsetzen. Es wäre allerdings kaum realisierbar, kurzfristig das gesamte Internet mit einer neuartigen Architektur zu ersetzen, weshalb in dieser Arbeit eine Schichtenarchitektur erforscht wird, welche aus mehreren, kleineren NDN-basierten Mobilfunk-Netzwerken (Domänen) besteht: Dabei werden die einzelnen Domänen untereinander mittels einem Chord Peer-to-peer Netzwerk verbunden, das auf bestehender Internetinfrastruktur betrieben werden kann. Um die Leistung dieser NDN & Chord Architektur zu evaluieren, wird sie in Form einer Netzwerksimulation im NS-3 Framework abgebildet, wobei auch diverse Charakteristika des existierenden Internets realistisch modelliert werden. Die Evaluation beinhaltet zudem ein Vergleich mit einer rein auf NDN basierenden Referenzarchitektur, was zum Vorschein gebracht hat, dass NDN & Chord eine valide und effiziente Alternative sein kann, wenn ein allumfassendes NDN Netz nicht realisierbar ist oder NDN Routing schwierigen Bedingungen unterliegt.

Acknowledgements

I would like to express my sincere gratitude:

To Prof. Dr. Burkhard Stiller, for making it possible for me to write a diploma thesis at CSG once more.

To Dr. Eryk Schiller, for sharing his valuable expertise and assisting this project around an exciting and relevant topic.

To my family and my friends for their hospitality, motivation and support throughout all of my studies since fall 2015.

Contents

Chapter 1

Introduction

The internet – as a global ecosystem of computer networks – has in the last decades evolved into the arguably most important means of communication used today. Its rapid emergence and growth would not have been possible without one of its underlying core networking techniques, *i.e.*, packet switching, which was developed over 50 years ago. Packet switching has crucially revolutionized computer networking by replacing its precursor technology – *i.e.*, circuit switching, in which for two network participants (*resp.* nodes) to exchange data, it was necessary to physically occupy a dedicated channel across the network. Instead, in packet switching any exchanged data between two hosts is encapsulated into small chunks (*i.e.*, packets), which are individually routed through the network. Thereby, as in real-world, each packet to be delivered is denoted with a source as well as destination address, which is meta information that allows nodes to decide to which next node to forward (*resp.* route) a packet to [31]. However, classical packet switching (as still used by today's internet), does not foresee metadata to be attached to packets revealing what kind of information or content is being transported.

Since the rise of the WWW (World Wide Web), the internet has become more and more a means for users to seek information and to consume digital content. For example, since smartphones are highly popular and mobile broadband subscriptions have become affordable, a growing number of users regularly consumes mass media content, such as news videos, during commuting. From a technical standpoint, the more users are accessing internet content, the more (streams of) packets have to be delivered from the content server to individual users. Especially in the case of mass media, a large number of users are accessing the same (popular) content, which implies that the same data is delivered from the server to multiple users over and over. This leads to an exhaustive amount of accumulated load put onto the network, which can be blamed on sparse packet metadata in the host-based paradigm: If packets held meta information about the transported content, it would allow for techniques to be deployed, in which popular content is delivered more efficiently, without straining network resources with redundant packet deliveries.

This is where Information Centric Networking (ICN) comes into play: Contrary to the host-centricity of the Internet Protocol (IP), ICN stands as a novel paradigm focusing on the transported information itself, *i.e.*, in terms of metadata, packets hold a name describing the contained data, instead of source and destination addresses [30].

1.1 Motivation

In recent years, several architectures were developed which implement the ICN paradigm and present themselves as future technologies to replace the traditional IP-based internet. Clearly, ICN implementations have the potential to improve the efficiency of increasingly important internet content delivery use cases. However, it is debatable for various reasons, whether ICN technologies are an optimal choice to globally replace the established IP-based ecosystem. For example, there is nowadays still a large number of use cases for which the host-based paradigm is ideal, *e.g.*, when establishing a connection to a remote server using SSH. At the same time, it is unlikely that an ICN architecture can be globally deployed within short-term; More likely, ICN-based networks will appear in terms of several smaller, independent domains.

In front of this background, this work proposes a multi-domain content delivery network architecture consisting of several mobile networks: Thereby, the mobile network domains operate on an ICN implementation, while to enable communication among different domains, a Distributed Hash Table (DHT) based Peer-to-Peer (P2P) network is used, which runs as an overlay network on top of traditional IP-based internet. With this approach, several independent mobile domains benefit from the advantages of ICN, while the exchange of data between foreign domains is efficiently carried out over existing internet infrastructure, *i.e.*, without the need for a globally deployed ICN network.

1.2 Thesis Outline

The following chapter 2 presents the thematic background as well as related work, in terms of introducing the relevant technologies and the state-of-science on which this thesis builds upon.

In the subsequent chapter 3, two different use case scenarios are presented, for which deploying ICN technology in mobile networks seems reasonable, when it comes to improving the efficiency of content *resp.* service response delivery.

In chapter 4, it follows a detailed description of the proposed architecture including its involved technologies, and how the architecture was implemented in terms of a network simulation in the simulation framework NS-3: This not only includes documentation of the simulation implementation details but also gives insight on involved measurements, models and assumptions which were applied to conduct network simulations as realistic as possible.

The results from conducting several simulation experiments to assess the performance of the proposed network architecture are presented chapter 5. This includes a comparison of the suggested architecture against a reference architecture which was simulated under the assumption of a globally deployed ICN infrastructure.

Finally, chapter 6 summarizes the contributions of this thesis and proposes potential future research.

Chapter 2

Background and Related Work

2.1 ICN: Information Centric Networking

Information Centric Networking is proposed as an approach to innovate today's internet architecture by promoting named data: Moving away from the classical host-based internet paradigm, ICN is information centric, *i.e.*, hosts in an ICN network request and deliver data according to names, instead of host addresses. As a consequence, in ICN, data is completely independent of storage, location and transportation, while it becomes a direct responsibility of ICN networks (*resp.*, participating nodes) themselves to ensure that data is efficiently delivered, *i.e.*, by applying data replication and caching techniques. This serves a vast number of information *resp.* content delivery applications, for which ICN stands as a promising alternative to the classical internet, while introducing more network efficiency and scalability. As globally, the demand for data continues to grow, maintaining network efficiency appears to be in everybody's interest. In front of this background the IETF (Internet Engineering Task Force) has originated the ICNRG (Information Centric Networking Research Group) in 2012, to promote research in the field of novel internet architectures implementing ICN [30].

2.1.1 ICN versus Content Delivery Networks

If we consider the main goal of ICN to optimize the efficiency of content delivery use cases, it seems reasonable to compare ICN to Content Delivery Networks (CDN) as a different technology having similar ambitions: The idea behind CDN is for content deliverers, such as music streaming services (*e.g.* Tidal), to distribute several edge servers (*resp.*, mirrors) geographically close to service users. This has the advantage that for user nodes to stream content, there is no need to connect all the way to the content provider's main server, but rather, the replicated content can be fetched from mirror servers that are deployed *e.g.* at some local ISP (Internet Service Provider) to which users are subscribed to. As opposed to ICN, CDN does not aim at replacing the current internet architecture, but rather stands as an approach to use the existing ecosystem to improve the QoE (Quality-of-Experience) for differently located groups of users. Another crucial difference to the ICN

paradigm is that CDNs rely on special contracts between content providers and network infrastructure operators, such as ISPs hosting edge servers: The contractual deployment of mirror servers can come at high costs, and it might be infeasible for content providers to deploy mirrors globally, such that only some network regions will benefit from faster content delivery [38].

In the ICN paradigm – on the other hand – data replication and caching are solved as a joint effort of the entire network, meaning that in comparison to CDNs, every node in the network allocates resources to automatically cache frequently requested data items and directly serves them to nearby data consumers. If ICN was used globally, content providers would not need to contractually deploy mirror edge servers and also, caching and efficient data delivery would be enabled between any kind of content source and its consumers (users), *i.e.*, not only for services actively investing in CDN infrastructure.

The following table summarizes the main differences between the ICN and CDN:

Paradigm	**ICN**	**CDN**
Used architecture	ICN as a replacement of classical internet	Deployed within existing internet infrastructure
Strategy for network efficiency improvement	Caching and data replication as a responsibility of every network node	Deployment of edge / mirror servers close to content consumers
Content provider independent?	Yes, all data cached and replicated regardless of origin	No, separate CDN servers for each provider
Involved cost	Implicit costs in terms of shared nodes' shared memory and bandwidth	Contractual costs between content providers and 3rd parties hosting mirror servers

Table 2.1: Comparison of ICN with CDN

2.1.2 NDN: Named Data Networking

Named Data Networking [2, 39] is likely the most prominent implementation part of the ICN architecture family. NDN originated from a preceding project, *i.e.* CCN (Content-Centric Networking), which was first introduced in 2006 [39]. The main idea behind NDN is to completely move away from the internet's classical host-based IP packet delivery, and instead implement fetching of named and secured data items. *Secured* refers to the fact, that in NDN, data originators cryptographically sign data packets, such that recipients can verify data integrity, regardless from which node in the network the data has been received.

Types of Messages and Network Nodes

Communication-wise, NDN operates by exchanging the following two types of messages [39]:

- **Interest packets** hold an identifier (*i.e.*, a name) and are used to inform the network about the demand to retrieve some named data item

- **Data packets** carry the name as well as content of requested data items

NDN networks feature the following network node types [39]:

- **Data producer** nodes provide a set of named contents (*i.e.*, data items)

- **Content consumer** nodes request data items by issuing interest packets

- **Intermediate nodes** either forward interest packets towards the upstream (*i.e.*, in the direction of producers), or, back-forward data replies from producers towards the downstream (*i.e.*, in the direction of consumers)

Naming Scheme

With regards to data item names, NDN proposes a hierarchical naming scheme, as used *e.g.* in unix-based file systems. For example, a mobile operating system security update could have the following name: `/google/security/android11/patch_7446.tar.gz`
If larger amounts of data have to be segmented into several packets, it is common in NDN to append sequence numbers to data names, such that data chunks can be individually retrieved and eventually constructed together again by content consumers. For example, a consumer might be interested in streaming a news video with name prefix `/srf/10vor10/2021-06-28.mp4` and append consecutive sequence numbers to it to construct interest names to retrieve individual file segments, *i.e.*: `[...]2021-06-28.mp4/1`, `[...]2021-06-28.mp4/2`, *etc.* [27, 39]

NDN Forwarder

One of the most important system parts of NDN is the NDN Forwarder, which resides in every node. Its reference implementation, which is realized as a community project, is called NFD (NDN Forwarding Daemon) [14]. In every NDN Forwarder there are three crucial data structures, namely Content Store (CS), Pending Interest Table (PIT) as well as Forwarding Information Base (FIB), which enable data caching, interest aggregation and routing in NDN. The responsibilities of these parts are explained in more detail in the following two NDN Forwarder operation examples:

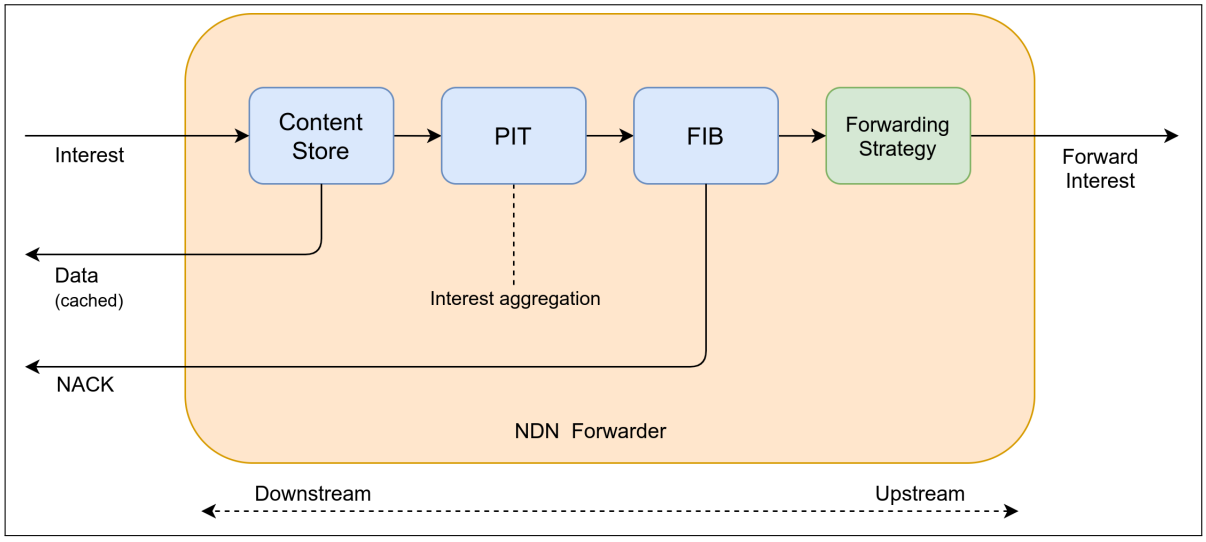

Figure 2.1: NDN Forwarder Upstream Operation

Let us assume that the NDN Forwarder depicted in the above figure is part of a node which receives an interest packet from another node in the downstream. It follows the following sequence of processing the interest packet [28, 39]:

1. The NDN Forwarder first checks, whether this node possibly has the requested data already cached in its Content Store: If so, the node can directly respond back with the cached data, *i.e.*, send the data back to the downstream (in the opposite direction from where the interest was received).

2. If however the data corresponding to the interest packet has not yet been cached on this node's CS, the NDN Forwarder writes the interest into the PIT table. This data structure is used to aggregate interests which could not be satisfied (*i.e.*, responded to with data) yet.

3. Next, the NDN Forwarder searches the Forwarding Information Base to determine, to which NDN interface(s) the interest should be forwarded to. The NDN FIB can be compared to the FIB in IP, but instead of mapping IP address prefixes to some next-hop interface, NDN FIB entries map name prefixes to one or several NDN interfaces. If the FIB does not contain any entry matching with the interest prefix, the interest cannot be forwarded and the Forwarder answers with a negative acknowledgement (`NACK`) back to the downstream.

4. If – however – the Forwarder has found a matching FIB entry, the interest will be forwarded to upstream nodes according to a configured forwarding strategy: With the *Best Route* strategy, interests are sent to the single next-hop interface with the lowest-cost (*resp.*, shortest path), while with the *Multicast* strategy, interests are sent out to all of the interfaces matching with the interest name prefix.

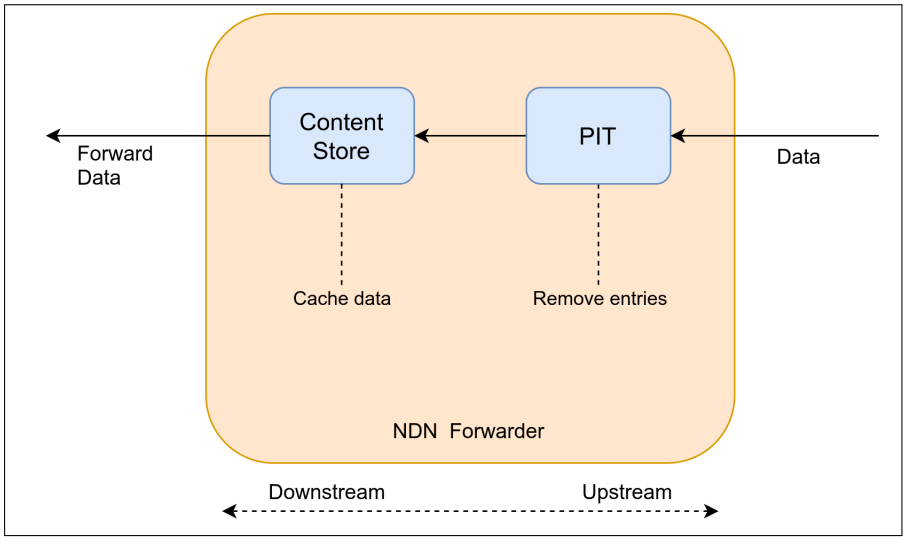

Figure 2.2: NDN Forwarder Downstream Operation

Continuing the preceding example, we now consider the NDN Forwarder operation (in the same node), if data arrives from the upstream, which satisfies the interest originally sent out [28, 39]:

1. When a data packet arrives from the upstream, the Forwarder queries the PIT to determine all interested downstream NDN interfaces, to which the data should be forwarded to. Eventually (*i.e.*, as soon as the data has been delivered), the PIT entries for the concerning interest will be discarded.

2. Before forwarding, the NDN Forwarder ensures to cache the data in its CS. This is to ensure, that for future incoming, identical interests, the (cached) data can be returned directly and efficiently from the current node, without the need to forward the interest further.

3. Finally, the data is forwarded to the set of downstream interfaces, which has been determined in the PIT (*cf.* step 1.).

2.2 SCN: Service Centric Networking

Besides the growing importance of content distribution, in recent years there has also been a clear trend for the internet to become increasingly service-based, in terms of evolving into a network of distributed, interactive applications. Examples of services can be named from many different fields, *e.g.*, high-definition video conferencing services, augmented, *resp.*, virtual reality, telematics (such as navigation applications) or various media processing services. While ICN implementations address the challenge of delivering static content (*resp.*, data) efficiently, they are not per default optimized for interactive services: Service Centric Networking (SCN) – on the other hand – is a variation of the ICN paradigm with the goals of optimizing the discovery and execution of services as well as the efficient delivery of service data [23]. It should be mentioned that while SCN promotes several features that make it optimized for services, SCN stands merely as an extension of ICN and is still based on the same principles defined in its origin paradigm. However, in contrast to ICN, SCN use cases benefit from the in-network caching of interactive service responses, rather than merely the caching of static data [19].

This is especially valuable in the case of service-based applications in which a lot of service users receive identical responses. Let us consider the example illustrated below, which is centered around a voice conferencing system: To realize that users can hear each other in real-time, there is a central conference (server), which mixes N individual digital input audio streams according to N participants' speech. In this instance, to retrieve the mixed audio, both participants, Alice and Bob issue an interest packet for a sequence of the audio stream (*i.e.*, `/mix/conf1/24`). The Conference Server answers with the requested digital audio sequence, which gets cached on intermediate nodes while moving towards the downstream. Bob's interest packet has to route through an additional node, and thus takes a longer delay to move towards the upstream: By the time the interest reaches the node with thick border, the response can be directly delivered from its CS cache, since Alice's identical interest has already been responded to [16].

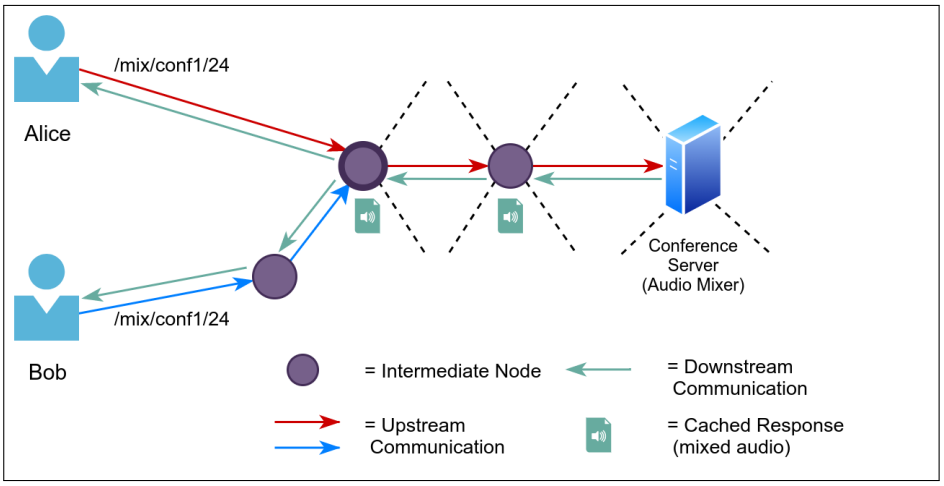

Figure 2.3: SCN Voice Conference Use Case Example

2.2.1 Layered Service Centric Networking (L-SCN)

In this section the focus is laid on *Layered Service Centric Networking* (L-SCN), an important related work which set the conceptual foundation for this thesis. L-SCN is an SCN implementation developed at the University of Bern and first presented in a 2017 IEEE publication by M. Gasparyan, T. Braun, and E. Schiller. Compared to existing implementations of the SCN paradigm, L-SCN introduces a novel approach to SCN routing. L-SCN is based on NDN and thus, complies with the ICN paradigm (*cf.* subsection 2.1.2): To conduct experiments and evaluate the architecture in a simulated environment, L-SCN has been implemented in ndnSIM 2.1 [27], an NS-3 based NDN simulation environment [19,20].

Layered Architecture

In comparison to existing SCN implementations, L-SCN presents an innovative network architecture, which describes two different communication layers, hence the name *Layered* SCN. This refers to the fact that in L-SCN, all participating nodes (this includes service providers as well as service consumer nodes) are clustered into several autonomous domains: Thus, L-SCN distinguishes between communication on an *intra-domain* layer, which describes the communication among nodes belonging to the same domain, as well as communication over an *inter-domain* layer, referring to the communication among nodes belonging to different domains [19, 20].

With regards to the construction of domains, the assumption is that the clustering of nodes is carried out according to node proximity, *i.e.*, in all domains, local nodes have great connectivity towards each other. *Proximity* does not necessarily refer to distance in the geographical sense, rather, the clustering process is assumed to group together nodes which can communicate to each other over high capacity links [19, 20].

Intra- and Inter-Domain Communication

What concerns intra-domain communication, L-SCN defines that nodes belonging to the same domain are directly reachable. Direct communication is however not possible in the case of nodes trying to reach out to services hosted in remote domains, since each domain is assumed to be self-contained (*resp.*, autonomous): To enable inter-domain communication, L-SCN selects in every domain at least one node, which gets elected as a *supernode*. Each supernode is connected to at least one other supernode of a foreign domain, and the requirement for nodes to be elected supernodes is stable connectivity to at least one foreign domain [19, 20].

The following figure illustrates intra- as well as inter-domain communication among three domains in L-SCN:

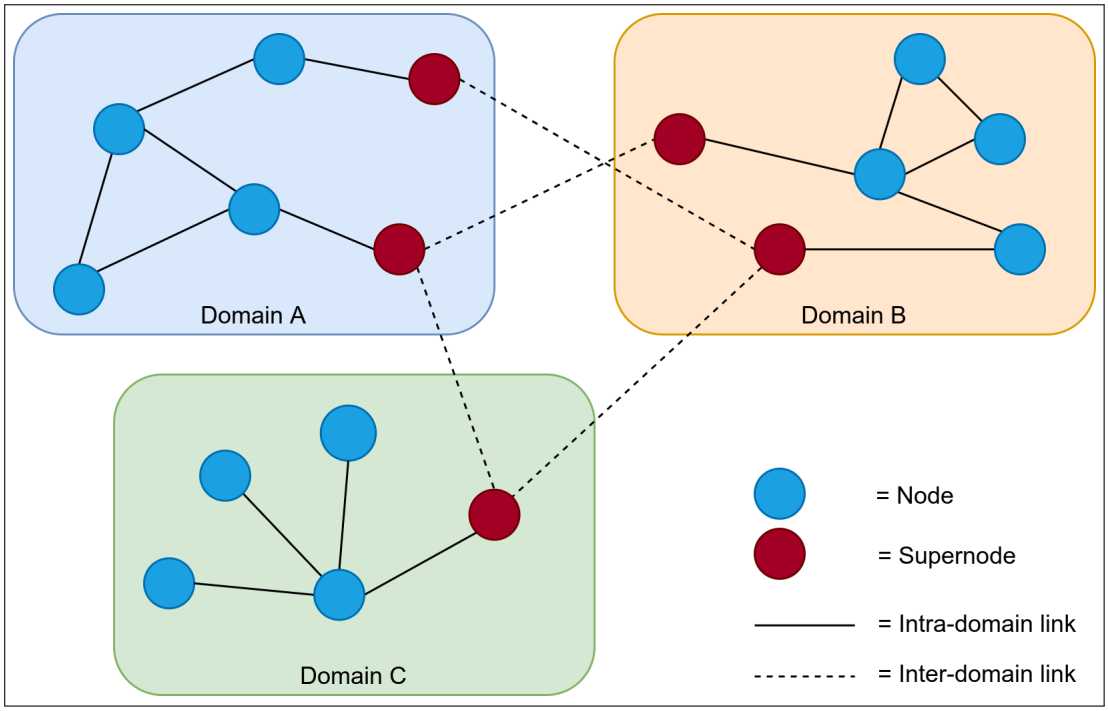

Figure 2.4: L-SCN Communication Among three Exemplary Domains

Intra- and Inter-Domain Service Discovery

From a Service Centric perspective, L-SCN supernodes are responsible for keeping track on all services as well as resources (*e.g.* computing power and memory available at nodes) offered in their corresponding domain. To retrieve this information, supernodes periodically send out Interest Information Messages (IIM) to all local nodes. Service providing nodes will respond to IIM packets with a Data Information Message (DIM) packet: DIM packets hold a Bloom Filter [15], *i.e.*, a bit field indicating the services hosted on the node. As soon as the local supernode has received DIMs all service providers, it merges all responses and forms a unified Bloom Filter, which represents the entirety of services provided in the domain. To make other domains aware of locally hosted services, supernodes then broadcast their domain-summarizing Bloom Filter to all neighbouring supernodes (*resp.*, domains) [19, 20].

It should be mentioned at this point, that during the service discovery process, L-SCN takes advantage of in-network caching defined in ICN and implemented, *e.g.*, in NDN. This brings along the following two advantages [19, 20]:

- Firstly, during service discovery, IIM and DIM packets are cached on intermediate nodes: If for example in some domain there are two (or more) supernodes and they broadcast identical IIM shortly after another, they might be served by intermediate nodes, which have the DIM packets already cached in their CS. Similarly, if some supernode A's IIM arrives at another local supernode B, B can either directly serve A with cached DIMs (if available), or, broadcast an IIM itself and finally forward

collected DIMs back to A. In both cases, the overall intra-domain network performance is improved, as IIM are prevented from being broadcast across the entire domain.

- Secondly, intermediate nodes between supernodes and service providers possess information on a fraction of the hosted services in the domain, since they cache forwarded DIM packets: If an incoming service request (interest) matches with information on a cached DIM, an intermediate node can efficiently forward the request towards the corresponding service provider node, instead of forwarding the interest to all neighbour nodes and thus flooding the upstream network.

2.3 Inter-Domain Communication as a Challenge

In general, current SCN as well as ICN implementations present solutions that work under the assumption that their proposed network architecture is deployed at the large scale, in terms of revolutionizing the current internet infrastructure and making all of its participants compliant with the new, proposed communication approach: However, performing such a transformation globally is not realistic, given the humongous scale, complex structure and ubiquitous use of today's internet. It is much more likely, that ICN and SCN networks will emerge in terms of several globally scattered zones *resp.*, domains. This – however – introduces several challenges, as firstly, inter-domain channels have to be set up, which usually have to operate over a different communication protocol than used within native ICN or SCN domains. Secondly, inter-domain communication can reveal itself as a performance bottleneck, if there is a lot of traffic among distant zones.

The following sections present how NDN and L-SCN address these challenges:

NDN at the Inter-Domain Level

Being proposed as a future internet architecture, NDN has the vision of transforming the entire existing internet into one global NDN network. For the reasons mentioned earlier, such a radical technology shift cannot be performed 'overnight' and is rather expected to happen gradually. At the same time, one could argue that there are today still a lot of internet-based use cases, *resp.*, applications working best with the host-based principle used in traditional internet architecture which raises the question whether a global NDN network is an optimal solution in every case.

To interconnect several NDN zones, the NDN project team provides the possibility to run NDN communication as an overlay network on top of traditional IP networks: If there are for example two distant NDN domains having no native NDN connectivity among them, they can be interconnected by establishing an IP-based connection among the NFDs (*cf.* section 2.1.2) part of interfaces in opposite domains: This is realized by encapsulating NDN packets into TCP or UDP packets and transporting them over the internet [9]. NDN as an IP overlay network (NDN-over-IP) is used in the NDN Testbed, a multi-domain NDN ecosystem used for research purposes [3].

While NDN-over-IP helps the deployment of NDN and enables domain-spanning NDN content delivery, NDN data encapsulation in IP packets leads to additional protocol overhead. At the same time, it can lead to an overwhelming amount of NDN control traffic, if a lot of content is requested from remote domains over inter-domain links.

L-SCN at the Inter-Domain Level

It was discussed in section 2.2.1 that the L-SCN architecture (in comparison to NDN) considers several service centric domains per design. Thereby, L-SCN reduces the amount

of service discovery protocol traffic on the inter-domain level, which is accomplished by using Bloom Filters as a lean data structure to broadcast service and resource availability information among the domains. This way, L-SCN can optimize communication at the inter-domain level, making it more efficient compared to intra-domain protocol communication [19, 20]. However, broadcasting information on the inter-domain level is still not optimal, especially if an L-SCN ecosystem is highly dynamic and involves a lot of domains offering a huge number of services and resources.

Being NDN-based, L-SCN can use NDN-over-IP to enable communication among disjoint domains. That is, if for example domain C in Figure 2.4 was disconnected from domains A and B per the lack of a global NDN network, the supernode of C could still communicate to A and B via an NDN-over-IP connection, *i.e.*, by establishing a connection using the IP addresses of the supernodes of domains A and B [19].

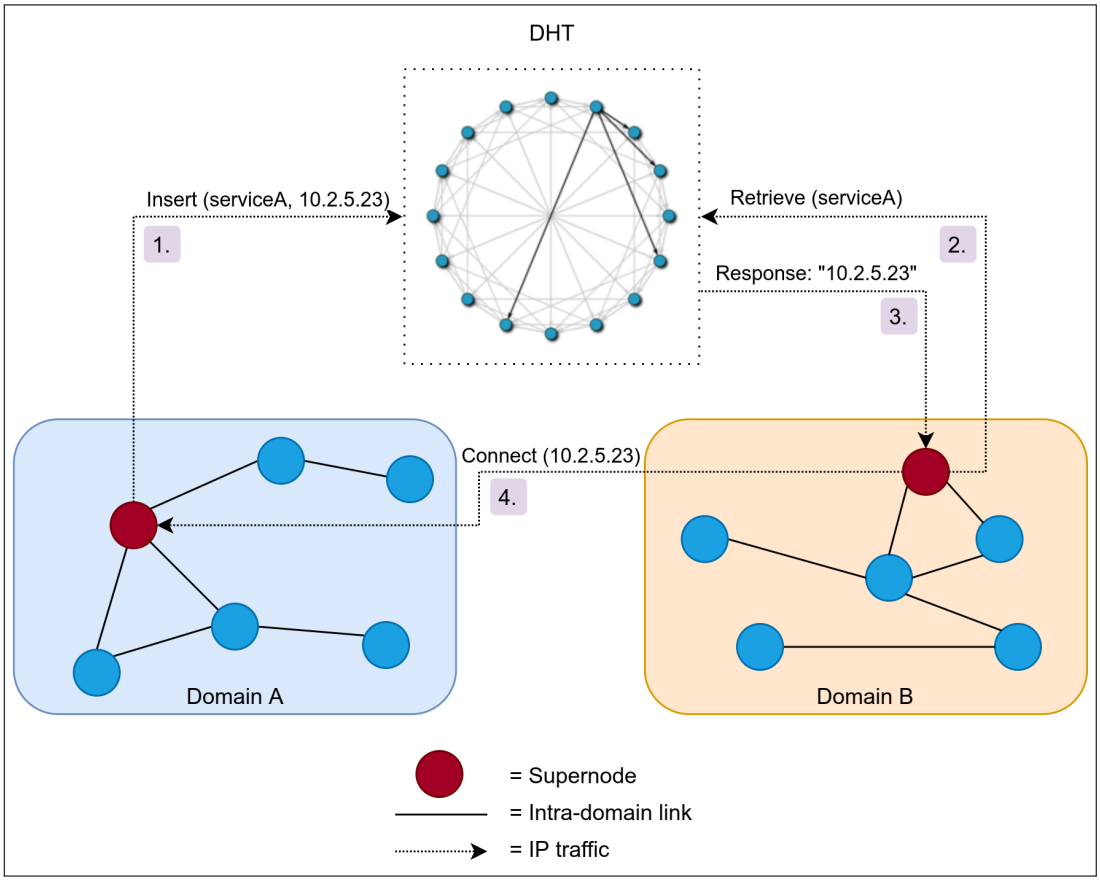

Figure 2.5: L-SCN Inter-Domain Communication using DHT

Certainly, the issue of excessive control traffic on the inter-domain level also exists if using NDN-over-IP to interconnect supernodes of individual zones. As a remedy, L-SCN presents a new approach to optimize inter-domain communication, by storing service availability information in a Distributed Hash Table (DHT, as introduced in more detail in the upcoming section): Simply put, a DHT is a data structure distributed equally among several internet hosts, which allows for the efficient storage and retrieval of data objects, *resp.*, information. In L-SCN, IP-capable supernodes can become members of a DHT,

building a DHT Peer to Peer (P2P) network. To expose the availability of local services, supernodes insert the relevant information in the DHT; Vice-versa, if supernodes want to lookup (*resp.*, locate) services offered in other zones, they can retrieve the directions towards the hosting domain from the DHT [19]:

In the example shown in Figure 2.5, L-SCN domains A and B are disjoint, *i.e.*, there is no native NDN connectivity among them. However, the supernodes of both domains are connected to the internet and are part of a DHT P2P network. The following sequence describes the illustrated L-SCN inter-domain service discovery with the help of a DHT [20]:

1. In domain A some node is the provider of `serviceA`. As soon as the supernode of A has discovered the service being provided within the zone, it expresses its availability to other zones: It does so by calling the `Insert` routine of the DHT and adding a tuple of [`service name`, `IP address`]. The IP address corresponds to the public address of the single supernode in domain A.

2. If in disjoint domain B, a request (interest) for `serviceA` arrives at the local supernode, supernode B calls the DHT's `Retrieve` routine with the service name as the key to search for, to look up the address of the supernode corresponding to the zone of `serviceA`.

3. The DHT replies to the supernode in B by returning the IP address of the supernode in domain A.

4. Finally, supernode B can connect to the domain hosting `serviceA`, *i.e.*, by establishing an NDN-over-IP connection with supernode A.

2.4 DHT-Based Peer-To-Peer Systems

2.4.1 Introduction

This section introduces IP-based Peer-To-Peer (P2P) overlay networks which are based on Distributed Hash Tables (DHT). Peer-To-Peer systems can be described as a crucial counter suggestion to traditional client-server systems: Instead of letting client nodes request data from a centralized server, P2P stands as a decentralized and distributed approach, in which every node (*resp.*, peer) acts both as a client as well as a server simultaneously. Thereby, all the peers part of the P2P overlay can communicate directly among each other, via the IP-based underlay network. Compared to classical client-server systems, in P2P, storage of data (*resp.*, information), peer lookup as well as data retrieval are responsibilities which are distributed among all participating nodes. This way, P2P systems can introduce a higher level of reliability, scalability and robustness. However, to share load equally among the peers, P2P systems require algorithms that fairly distribute load among all participating nodes: Here, the term *load* can refer to the following aspects [18]:

- *Request load* results on peers which have to answer queries for locally stored objects (*i.e.*, data items), *w.r.t.* exchanged messages, data and spent resources (computational power) to answer queries
- *Object load* is caused according to the size and popularity of requested objects present in the system
- *Routing load* results when nodes in-between requesting and serving peers are forwarding (routing) query traffic

The upcoming section presents how DHT-based overlay (*resp.*, P2P) networks address the challenge of load balancing.

2.4.2 Distributed Hash Table Overlay Networks

In general, DHT overlay networks follow a design which allows for well-balanced request, object and routing load. That is, all of the peers have to respond to a comparable amount of queries, data objects are requested at a similar popularity, while the flow of requests is balanced equally. This is accomplished by the following high-level overlay architecture [18]:

Identifier Space

A DHT overlay consists of a set of peers (*i.e.*, nodes), while each peer features a unique identifier (*i.e.,* an IP address) that belongs to a certain identifier space. To enable communication among the peers, links (*i.e.*, connections) are set up among pairs of participating peers. One of the core principles of the overlay is the even, collaborative storage of objects

among the peers, while each peer has to store a fraction of the total amount of objects: This is realized by dividing the identifier space into a series of non-overlapping identifier ranges, which are assigned to participating peers. All of the objects in the network have an identifier (*i.e.*, a key), which follows the same identifier space as the peers: This way, objects (according to their identifier) can be assigned to be stored on the peer which is responsible for the corresponding identifier range. If some peer queries a certain object from the DHT, a routing function ensures to route the requests towards the node which is responsible for the identifier range to which the object belongs to. Depending on the routing algorithm in-place and the shape of the network *w.r.t.* established links among the peers, the request possibly traverses a series of nodes until the serving peer is reached. In most cases, DHT utilize greedy routing algorithms, meaning that peers greedily forward queries to the some known neighbour which is the closest to the queried object in terms of identifier space. To achieve a fair object load among the peers, node and object identifiers (*i.e.*, keys) must be distributed equally over the identifier space. This can be achieved by an approach called *namespace balancing*, which incorporates the idea of hashing the identifiers of peers and objects (*i.e.*, IP addresses, and object names) and using hashed identifiers to equally assign objects to nodes in the identifier space. [18].

Routing Tables

Such that in DHT overlay networks, peers can route object queries towards their destination, each peer holds a routing table, in which it stores outgoing links to its directly reachable neighbour nodes. When a routing algorithm at some node has to decide to which node to forward a lookup request to, it chooses the neighbour in the routing table, which identifier-space-wise is closest to the destination. Routing tables are essential for the lookup mechanism to function properly, which is why DHTs implement mechanisms to ensure that routing tables remain up-to-date: When an obsolete (*i.e.*, inactive) neighbour node entry is detected, the routing table is fixed by replacing the entry with a different node which inherits the responsibilities from the now inactive peer (*i.e.*, according to the rules of identifier space management in-place) [18].

2.4.3 Chord DHT Overlay

This section introduces Chord, a state-of-the art DHT overlay developed at MIT and introduced in 2001 [32].

Identifier Ring

In Chord, each data object and peer is assigned an identifier consisting of m bits. To enable namespace balancing, Chord computes m-bit identifiers by hashing data object names, and peers' IP addresses using the SHA-1 algorithm. This way, peers and objects can be uniformly distributed in a ring-shaped identifier space, while due to the identifier size of m bits, the identifier ring can hold up to 2^m different addresses [18, 32].

Chord defines that each peer must know its direct successor as well as predecessor peer in the identifier ring: While the predecessor is a peer's next peer in counter-clockwise direction in the identifier space, the successor refers to the upcoming peer in clockwise direction. As mentioned, object names adhere to the same identifier space as the peers, thus, the identifier ring is used to determine the peer which is responsible for the storage of some object. Chord organizes this by assigning objects to their successor peer: If *e.g.*, there is an identifier sequence (in clockwise direction) of A, B, C in the identifier ring, while A and C are peers and B is an object, it is the responsibility of C to store object B. Thus, if a peer receives a query for an object which is not stored locally, the protocol will let the peer forward the request either in the direction of the predecessor, or, successor peer, depending on the position of the object in the identifier ring [18, 32].

Finger Tables

As mentioned, in Chord every peers must at least know its successor and predecessor peers. If – however – these were the the only known peers, looking up objects could potentially be very inefficient, as requests would in the worst case have to surpass the entire identifier ring, until reaching the direct successor of some object; Complexity-wise, this would correspond to a linear search at $O(n)$. To improve that, Chord introduces the concept of Finger Tables, which are a special type of routing tables residing in every peer [18, 32]:

Finger tables can contain up to m routing entries, while m is the fixed size of the identifiers in bits. The first entry of a finger table always corresponds to the peer's immediate successor. The remaining $m-1$ entries correspond to different peers of the network, which are derived according to the following formula (n denotes the current peer, i denotes the number of the entry in n's finger table): $i = \texttt{successor}((n + 2^{i-1}) \ mod \ 2^m))$ [32].

Keeping a set of m routes in every finger table ensures that every peer knows m 'shortcuts' to other peers across the entire identifier space: Subsequently, when a peer wants to lookup an object o, it searches in its finger table for the peer which is closest to the successor peer of o and forwards the request towards it. By using this routing strategy, it is possible to prove that the lookup complexity can be improved to $O(\log n)$ [32].

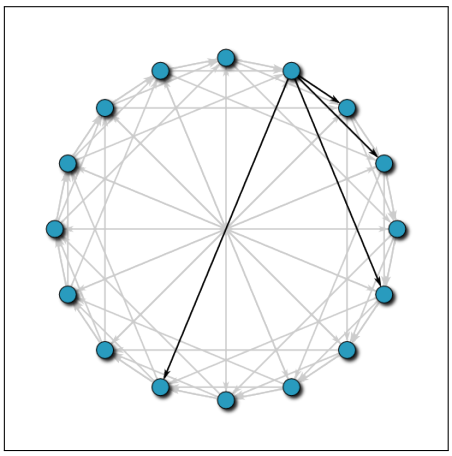

Figure 2.6: Chord Finger Table [33]

The above figure depicts the finger table of a peer in a Chord network of size = 16: While the first 'finger' points to the immediate successor in clockwise direction, the second, third and fourth fingers point to successor peers according to powers of 2 going into the clockwise direction of the identifier ring (starting at the current peer) [32].

The figure below illustrates the routing process of some peer A requesting an object from peer B:

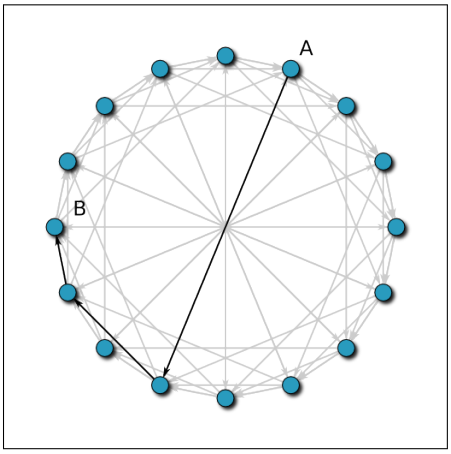

Figure 2.7: Chord Finger Table [34]

Due to finger-table based routing, the Chord routing process can improve peer lookup significantly, reducing the number of required hops until reaching the serving peer. Thereby, at each hop towards the destination, the distance is effectively divided in half [32]. However, since Chord operates as an overlay on top of the internet, additional routing overhead might occur: Even if Finger Table routing effectively reduces routing effort, one hop between two overlay peers might correspond to multiple hops in terms of intermediate nodes in the underlay.

Stabilization Mechanism

The Chord protocol runs a stabilization routine in the background, which ensures, that the lookup process always functions properly, even in the case of peer churn (*i.e.*, when peers join or leave the overlay). For example, if a peer leaves the network (or gets inactive), the finger tables of all peers pointing at that peer have to be updated, such that they do not point at an absent peer. The same is true for the opposite case: If a peer Y joins the Chord ring in between a predecessor X and a successor Z, Y inherits a part of the objects for which Z has been responsible. Subsequently, all peers' fingers have to be updated, such that the objects, which now are under the responsibility of Y, can be retrieved.

Chapter 3

Use Case Scenarios

In this chapter, two different use case scenarios are presented, which are centered around the idea of efficient content, *resp.* data retrieval through mobile networks. Both scenarios have the common property that there is a high probability for the user devices part of the mobile network to (independently) request a large number of identical data items, *resp.* information. Thus, it is highly beneficial to put into place ICN-based in-network caching techniques (*cf.* section 2.1), as realized by *e.g.* NDN, to ensure that popular data is available in close proximity to requesting devices.

3.1 Mobile OS Patch Distribution

In the past two decades, the emergence of internet-powered mobile devices, such as smartphones, has lead to a massive growth of attack surface when it comes to security loopholes in mobile operating systems (OS). This is especially problematic for highly popular mobile OSs, since attackers naturally tend to choose prominent platforms on which to discover and exploit security holes, to cause the greatest possible damage. For the globally most widely used mobile OS, *i.e.*, Android, this has in the past led to a vast number of discovered and fixed vulnerabilities [10].

Let us in front of this background consider a scenario, in which recently, there has been a security gap discovered affecting the Android OS, which allows attackers to inject malicious code (*i.e.*, malware) into mobile devices: On affected devices (*e.g.*, tablets), the malware circumvents the system's hardware control mechanisms, such that attackers can have full access to *e.g.*, cameras, microphones and various sensors. Of course, this presents a highly severe intrusion, breaching the privacy of potentially billions of Android users. With that being said, it is both in the interest of the OS maintainer (*resp.*, the Android Security team at Google [11]) as well as all potentially affected users, for this security gap to be fixed as soon as possible.

Continuing this scenario, let us now assume that the Android Security team has fixed the vulnerability and now must ensure that an OS patch file be distributed and installed on all Android devices, such that attackers will be hindered from further misusing the

security gap: From a data distribution perspective, this is a situation of a vast number of scattered mobile nodes requesting the same data item from some serving host residing at one specific location. For the network between the mobile devices and the server offering the security patch, this clearly is highly inefficient, as per the host-based design of the internet, each device must establish a separate connection with the server to download the file of interest.

It would be far more efficient, if an ICN implementation was introduced, which would automatically sense that identical data is requested by many nodes, such that the patch can be cached on intermediate network nodes making the file available closer to interested devices.

The following figure depicts how this scenario could benefit from the NDN paradigm:

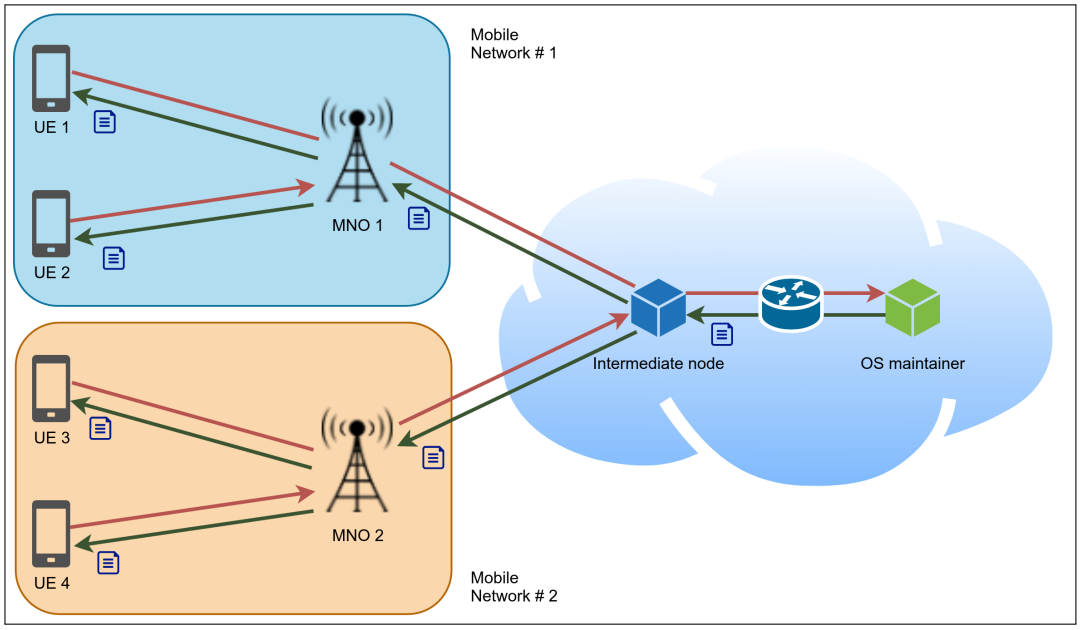

Figure 3.1: Mobile OS Patch Retrieval Using NDN

Let us consider that the mobile devices (*resp.*, UEs (User Equipment)) request the security patch (blue file symbol) in order of their consecutive numbering:

`UE 1` is the first device to express interest for the patch to the network, therefore its request is routed all the way upstream towards the OS maintainer (indicated by the red arrow). As soon as the OS maintainer sends out the patch file (indicated by the green arrows), intermediate hosts (*e.g.*, `MNO 1` (Mobile Network Operator) as well as the `Intermediate Node`) cache the file for future requests from the downstream. By the time `UE 2` requests the patch, it will already be cached on `MNO 1`, thus, it can directly and efficiently be retrieved from there.

As soon as `UE 3` requests the patch, it will be returned from the cache on the `Intermediate node`, since from the perspective of `Mobile Network # 2`, this is the closest node in the upstream that holds the file cached. Finally, the situation for `UE 4` is the same as for `UE 2`, as it can retrieve the patch directly from the local `MNO`.

3.2 Image Recognition Cloud Application

The previous section presented a use case scenario in which ICN improves the efficiency of the delivery of static data. This refers to the fact that in the case of retrieving a security patch file, a huge number of devices are requesting the exact same data object. In this use case scenario – however – we consider a cloud-based mobile application, in which mobile devices request and receive dynamically created service responses, which can be optimized by the use of Service Centric Networking implementations (*cf.* section 2.2). The considered example of such an application is Google Lens [1], which incorporates an image recognition based cloud service:

With Google Lens, users can simply point their smartphone's camera to objects and retrieve various types of information related to them. This is driven by neural-network-based visual analysis techniques and allows for the following examples of features [1]:

- Exploring places of interests (*e.g.*, towers in London)
- Scanning and translating text from a foreign language
- Identification of animals and plants
- Automatic detection of problems from various disciplines (*e.g.*, chemistry homework) and provision of step-by-step solutions

In the following Google Lens inspired scenarios we consider the use of a cloud service powered image recognition application during a sightseeing tour:

1. Let us assume we are visiting Zurich as tourists and would like to explore historical buildings. We find ourselves for example on the Grossmünsterplatz and would like to retrieve information and insights about the church building located in this area. We do so by starting an image recognition application on our smartphone and pointing the camera at the two towers of the Grossmünster church. The assumption is that the application is equipped with a machine learning based object classification feature, which manages to classify the object in the taken image as a church building, *i.e.*, natively on the device. This information, together with the local GPS coordinates determined by the smartphone is sent as a tuple of parameters to the cloud service: `<"Church", (47.370, 8.543)>`

   Subsequently, the service request is transmitted over the mobile network to the cloud service backend, which – based on the recognized object class as well as the sensed coordinates – determines that the Grossmünster must be the object of interest. This allows for compiling a series of information and media which the service delivers as a response, *e.g.*, a summary on the historical significance of the building as well as pictures from the inside.

2. We continue our virtual sightseeing tour in Zurich and now find ourselves at Unionstrasse, in front of the former home of Albert Einstein. The house is decorated with a memorial plate having engraved on it a text in German, which we would like to

scan with the application and retrieve an English translation: Here, we assume that the mobile application's image recognition feature has the ability to perform OCR (*i.e.*, Optical Character Recognition) and thus can convert the image taken into a string of the memorial plate text in German. The recognized text is then transferred as a parameter to the cloud service: `<"Hier wohnte von 1896-1900 der [...]">`

Once the cloud service has received the memorial plate text, it will translate it text to English using machine translation technology. The translated text is finally transferred to us as users of the mobile application.

The illustration below indicates the two described image recognition cloud service use cases. The green arrows correspond to service requests issued by mobile devices and forwarded towards the cloud service, while the red arrows indicate the transmission of service responses from the cloud service directed towards mobile devices. The document symbols indicate for both use cases the possibility to cache service responses, for example at a local MNO in proximity to the service users.

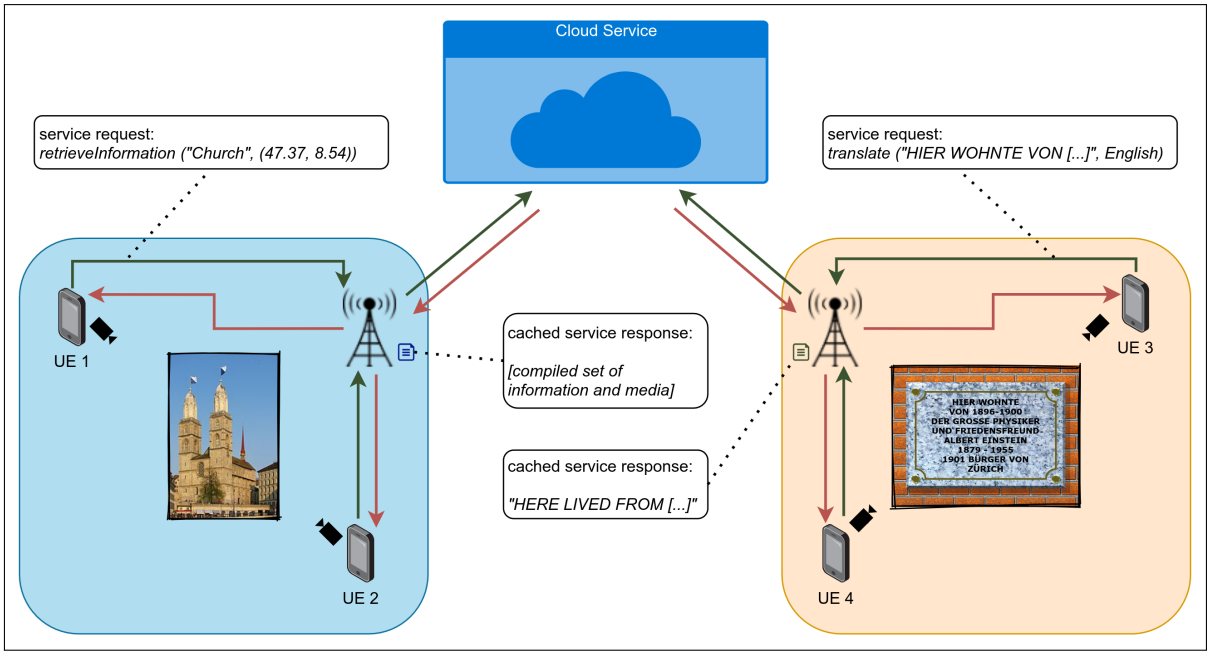

Figure 3.2: Image Recognition Cloud Service Powered by SCN

Clearly, in both examples above, it seems reasonable to introduce in-network caching of service responses, as provided by SCN implementations:

In the first case, it is very likely that a lot of tourists visit the same place and would like to retrieve the same information by using the mobile application. Thus, the cloud service would receive a lot of identical or similar requests (interests) with the object classification (*i.e.*, "Church") as well as local position coordinates. If an SCN implementation was deployed in the network between service users and the cloud service, it would be possible to cache the compiled information and media data on the Grossmünster on local intermediate nodes: The same is true for the translation use case, as a lot of people will scan the same

memorial plate text and request a translation for it: Using SCN, the translated text can be automatically cached in close proximity to the service users.

In both cases, SCN can improve the performance of both the network as well as the cloud service: As in ICN, caching data closer to users prevents from a lot of network-load heavy data exchange round trips between service consumers and service providers. At the same time, the cloud service is discharged from an extensive load of service requests leading to high computational processing loads: For example, some foreign language text has to be translated only once by the service, further translation requests can directly be served from caching nodes closer to the requesting device.

Chapter 4

Architecture Design and Implementation

This chapter introduces the architecture proposed and implemented in the course of this work. In the second chapter, it was discussed in section 2.3 that globally replacing the existing internet with an ICN-based architecture can be described as an unrealistic objective for various reasons. Like suggested by L-SCN [19, 20], in this project therefore a layered architecture is considered, which distinguishes between an intra- and inter-domain communication level:

On the intra-domain level, we define several autonomous zones in terms of LTE-based *last-mile* mobile networks, which operate on NDN. Each zone follows the typical hierarchical structure of LTE while only the hierarchically superior nodes are responsible for caching data requested from within the zone. To enable communication at the inter-domain level, we elect in each mobile zone one supernode, namely at the highest level *w.r.t.* the LTE hierarchy. Zone supernodes are connected to the internet and are involved in a Chord DHT ring, which is spanned among all the domains and is used as a native data object store to enable data retrieval among different domains: If for example some mobile node in a last-mile zone requests a specific data item, which is not (yet) cached in the local zone, its NDN interest will be forwarded into the direction of the zone's supernode, while the supernode translates the interest into a data object retrieval on the Chord DHT.

Using NDN within last-mile zones is expected to enable automatic in-network caching of data frequently requested by users' mobile devices. At the same time, deploying a Chord DHT on the inter-domain level should allow for a well-structured and efficient strategy of organizing data storage and delivery among foreign domains. Since Chord operates over the classical, IP-based internet, already established infrastructure is used for inter-domain communication, hence, a global-scale NDN network is not required.

The following figure depicts the proposed architecture:

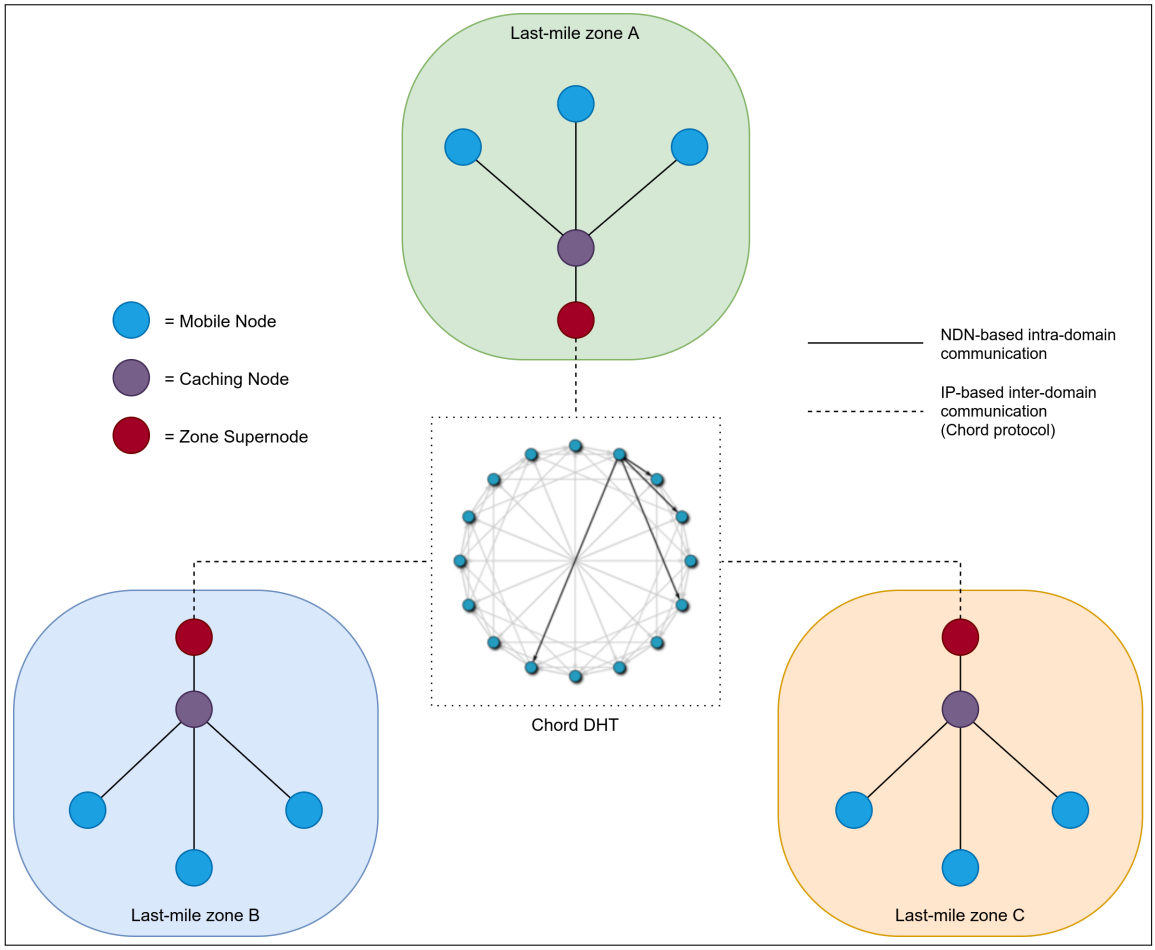

Figure 4.1: Proposed NDN & Chord Architecture

The exemplary topology shown in the above architecture illustration consists of three NDN last-mile mobile networks (*resp.*, zones), while there are three data consuming mobile nodes associated with each of the zones. Next to mobile nodes, each network features an intermediate node responsible for caching as well as a zone supernode, which is part of the Chord DHT object store. Communication-technology-wise, all nodes involved communicate over the NDN protocol exclusively, with the exception of the supernodes, which are also part of an IP-based network to enable inter-domain data delivery using the Chord DHT overlay.

To conduct evaluation experiments with the proposed layered NDN & Chord architecture, it was implemented in terms of a network simulation using the environment presented in the upcoming section 4.1.

The subsequent section 4.2 addresses the modelling of LTE-based last-mile mobile networks.

In section 4.3 the bridging of ICN & DHT in terms of the translation between NDN at the intra-domain as well as Chord at the inter-domain level is presented.

For properly studying the benefits of the proposed architecture the aim was to simulate real-world internet characteristics at the small scale: To accomplish that the focus is laid

on two main properties, namely data object popularity, as well as network connectivity of the internet at the inter-operator level. The former is addressed in section 4.4 which gives insight on how mobile consumer nodes' behaviour is simulated to reflect real-world data demand patterns, while the latter is discussed in section 4.5, which lays the focus on the modelling of a transit network enabling communication on the inter-domain level.

To make it possible to compare the proposed NDN & Chord architecture with regular NDN networks in terms of simulation experiments, a layered reference architecture was defined and implemented, which uses NDN as the only communication protocol, as described in section 4.6.

The concluding section 4.7 includes documentation of the two main developed programs enabling network simulation experiments with the proposed NDN & Chord as well as the plain NDN reference architecture.

4.1 Simulation Environment

4.1.1 NS-3 Network Simulator

The practical part of this project, *i.e.*, setting up the planned architecture in terms of a simulation and conducting experiments, was carried out within the NS-3 simulation framework: NS-3 is a discrete-event network simulator written in the C++ programming language. Since 2006, NS-3 is developed and actively maintained in terms of an open-source project. *Discrete-event* describes a class of simulators in which during the execution of simulations, each event (*e.g.*, some node starting a data transmission) is coupled to a specific moment in (simulation) time, while during execution, the simulator can efficiently hop from one event execution to the upcoming, in the temporal order of the scheduled events [6].

While it is possible to simulate a variety of different networking technologies, NS-3 defines the following crucial abstractions, which refer to the same concepts in every simulation scenario [6]:

- *Nodes* are the basic abstraction of devices part of a simulated NS-3 network.
- *Applications* can be seen as simulated software, which is installed on Nodes to trigger any kind of action to be carried out (*e.g.*, responding to incoming data requests)
- *Channels* are the representation of physical media over which data flows. A Channel can *e.g.* be configured to simulate the performance metrics of an Ethernet connection.
- *Net Devices* model the combination of networking hardware as well as driver software allowing Nodes to communicate among each other over a specific Channel.
- *Helpers* refer to a common type of NS-3 classes, which improve the efficiency of setting up large and complex topologies by automating the task of installing Net Devices on Nodes, and Net Devices to Channels, *etc.*

Using the above abstractions, it is possible to set up simulation scenarios in terms of single-file C++ scripts, which directly interact with the NS-3 core source code [6].

The NS-3 source code consists of several software libraries, called *modules*. Each module contains one or more *models*, which represent real-world objects or networking protocols. Over the years, several modules have been made a permanent component of the main NS-3 source code, *e.g.* the LTE (Long Term Evolution) module with its components LTE model as well as EPC (Evolved Packet Core) model [5]. It is possible to import modules into any NS-3 code base which are either new or are not yet part of the main module library. This – however – is only possible if the imported module as well as the version of the used NS-3 code base are compatible with each other. To ensure compatibility with the non-standard modules used, this project is carried out in NS-3 version 3.29, which was released in September 2018 [8].

4.1.2 ndnSIM: NS-3 Based NDN Simulator

All simulations configured and carried out in this project involving NDN networks are based on ndnSIM, an open-source, NS-3 based NDN simulator module originating from the University of California, Los Angeles. The first version of ndnSIM was introduced in 2012, while in 2015, a heavily restructured second release (*i.e.*, ndnSIM 2.0) was introduced [27, 28]. For compatibility reasons, in this project ndnSIM 2.7 is used, which was released in February 2019.

Compared to the first release, ndnSIM 2.0< versions directly integrate with the reference source code of the NDN Forwarding Daemon (NFD, *cf.* section 2.1.2) instead of re-implementing basic NDN routines. The same is true for NFD's auxiliary library "NDN C++ library with eXperimental eXtensions" (ndn-cxx, [35]), which provides several functions around NDN packet encoding as well as security mechanisms [27, 28].

Both NFD as well as ndn-cxx are contained in the NS-3 ndnSIM module library, allowing for realistic NDN simulations in NS-3. ndnSIM features a helper class (*i.e.*, `ndn::StackHelper`), with which it is possible to conveniently install the complete NDN protocol stack on NS-3 Nodes [27, 28]. Thereby, Nodes are equipped with the basic NDN data structures and their corresponding functionalities, *i.e.*, Content Store, Pending Interest Table as well as Forwarding Information Base (*cf.* section 2.1.2).

ndnSIM-specific Applications

As stated in subsection 4.1.1, next to Nodes, Channels and Net Devices, NS-3 simulation scenarios rely on Applications installed on Nodes, to actually create network activity (*e.g.*, a packet exchange). This is also the case with ndnSIM, in which specific applications allow for generating NDN interest *resp.* data flows among Nodes. The following list indicates three default ndnSIM applications [27, 28], of which the latter two were modified to simulate and evaluate the proposed NDN & Chord architecture (*cf.* section 4.3, section 4.4):

- `ConsumerCbr` is an application which lets consumer nodes send out a series of Interest packets at a constant frequency (*resp.*, bit rate)
- `ConsumerZipfMandelbrot` is a derivation of `ConsumerCbr` with the difference of generating Interests for data objects according to the Zipf-Mandelbrot distribution (*cf.* section 4.4)
- `Producer` is a basic application for data producing nodes which can be configured with a specific name prefix to produce data for; Upon incoming interest packets with matching name prefix, `Producer` lets Nodes respond with Data packets holding virtual payload

Forwarding Strategies and Routing

Per its integration with the NFD library, ndnSIM allows users to specify the forwarding strategy to be used, when intermediate nodes have to forward Interest packets towards the upstream. The following two strategies are used in the scope of this work [27, 28]:

- The *Best Route* strategy ensures that Interest packets are forwarded to the (single) matching upstream node with the lowest cost *w.r.t.* routing
- The *Multicast* strategy enforces Interests to be broadcast to all upstream nodes which are recorded for the FIB entry matching with the Interest prefix

For the Best Route strategy to function properly, it is necessary to populate complete and correct FIB routing tables among all nodes, such that each forwarding node can determine not only the correct upstream node, but at the same time the node with lowest routing costs to send an Interest to. With the Best Route strategy, incomplete FIBs can lead to nodes sending back negative acknowledgements (`NACK`) to the downstream due to a missing (lowest-cost) route, while the Interest is being discarded and not forwarded further. The automatic population of integral routing tables can be achieved by a routing helper class part of ndnSIM (*i.e.*, `GlobalRoutingHelper`) [27, 28]. Compared to Best Route, the Multicast strategy is less critical *w.r.t.* to FIB completeness, as nodes configured to forward according to Multicast will forward Interests to all upstream nodes, regardless of whether it is possible to gain routing cost information.

Content Store Configuration

The API provided by the `ndn::StackHelper` class allows for configuring the Content Store (CS) which is part of every NDN node's Forwarding Daemon (NFD). Depending on the chosen CS parameters, the simulated NDN network will behave differently *w.r.t.* data item caching, which can have a strong influence on the network performance. The following two CS properties can be configured [27, 28]:

- `CsSize` allows for setting the maximum size of the Content Store, in number of packets
- `Policy` refers to the chosen cache replacement policy (*i.e.*, `LRU` or `FIFO`)

`LRU` (Least Recently Used) ensures to discard those data items first from the cache which were not requested for the longest amount of time. `FIFO` – on the other hand – does not take request frequency into account but rather discards data items in the same order as they were added to the CS [27, 28].

NDN simulations part of this project use the `LRU` strategy per default, such that frequently requested data items can be efficiently cached in close proximity to requesting nodes.

4.1.3 ns-3-chord: Chord/DHash DHT in NS-3

Chord/DHash DHT (`ns-3-chord`) is a library for simulating Chord in NS-3, which was introduced by Harjot Gill from the University of Pennsylvania in 2009 [21]. In this project, `ns-3-chord` is used to simulate Chord as a Peer-to-peer system to natively store data objects *resp.* items. `ns-3-chord` is not part of the standard set of NS-3 modules, but to my best knowledge, it is the only Chord simulation module available for NS-3. As there is no descriptive documentation available for `ns-3-chord`, this section only introduces it in terms of core characteristics as well as basic setup and usage requirements.

Core Characteristics and Usage

In `ns-3-chord`, Chord protocol messages (*e.g.* object lookups) are exchanged over the UDP protocol, while for transferring data objects, TCP is used [21].

For `ns-3-chord` to function in an NS-3 simulation, the minimum requirement is a set of interconnected Nodes, on which the IPv4 stack is installed: In NS-3, this can be accomplished automatically using a helper class called `InternetStackHelper` [5]. If this requirement is fulfilled, it is possible to install the `ns-3-chord` helper class `ChordIpv4Helper`, which automatically sets up the Chord overlay application (*i.e.*, `ChordIpv4`) on chosen Nodes. This allows for interaction with the following high-level `ns-3-chord` routines (only the ones used in this project are introduced) [21]:

- The `InsertVNode` routine allows for defining Nodes as Chord Virtual Nodes (`VNode`). The first `VNode` which is inserted is a so-called bootstrap-node, which will be contacted by every later joining `VNode` to receive initial information about the already-existing Chord ring. The `InsertVNode` routine also triggers a re-distribution of the key-space, to ensure that every `VNode` is responsible for an equal range of keys (*resp.*, objects, *cf.* subsection 2.4.3).

- The `Insert` routine is used to insert data items (*resp.,*) objects into an established Chord ring during a simulation. As introduced earlier, in Chord all data objects possess a key, which is also simulated alike in `ns-3-chord`: Data objects can be inserted from any active `VNode` in terms of a key-value pair (value refers to data payload), while the key gets digested (*resp.*, hashed) using the SHA-1 encryption algorithm. Using the hashed key, `ns-3-chord` determines the owner `VNode` of the given key and then sets up a TCP connection to it to transfer the new data object.

- With the `Retrieve` routine, the `ns-3-chord` overlay can be invoked to retrieve a data object (according to its corresponding key) which has previously been inserted into the simulated DHT ring. This is a two-fold process that first involves the looking up the object key in question: This (as in the `Insert` process) is accomplished by hashing the key using SHA-1 and letting `ns-3-chord` determine the owner `VNode` responsible for the object. Finally – once the object owner has been determined – the data is transferred to the requesting `VNode` over TCP.

All the above presented `ns-3-chord` routines, as well as further functions (not used in this project) feature `Callback` functions, which in NS-3 are used to inform about certain events across several modules *resp.*, applications or protocols [4]. The `Retrieve` routine – for example – features two `Callbacks`, one for notifying the successful retrieval of some data object (*i.e.*, `RetrieveSuccessCallback`), and another for reporting a object retrieval failure (*i.e.*, `RetrieveFailureCallback`) for the case no object owner could be determined, or, the transmission had failed [21]. The `ns-3-chord Callback` methods for the `Retrieve` routine is especially important for the bridging between NDN and Chord, which is presented in section 4.3.

4.2 Configuration of Last-Mile Zone Networks

This section addresses the simulation NDN-based last-mile mobile networks that constitute the intra-domain layer of the proposed NDN & Chord architecture. The initial idea for this project was to simulate last-mile networks using the LTE simulation module [5] part of NS-3, which would have provided the most realistic LTE network simulation possible. However, for the reasons mentioned in the upcoming section, this could unfortunately not be realized, which is why a custom mobile network simulation was designed, as presented in subsection 4.2.3.

4.2.1 LTE-EPC Simulation in NS-3

LTE-EPC, *resp.*, LENA (**L**TE **E**PC **N**etwork Simul**A**tor), is the default NS-3 LTE simulation module consisting of the components LTE as well as EPC model: The LTE model simulates the LTE radio protocol stack, which is installed on UE (User Equipment) and eNodeB (Evolved Node B) base station Nodes to enable over-the-air connectivity. The EPC model – on the other hand – ensures the simulation of end-to-end communication in terms of IP packet exchange within LTE networks [5].

The following illustration gives an overview over the NS-3 LTE-EPC module architecture, indicating its different node, interface and link types:

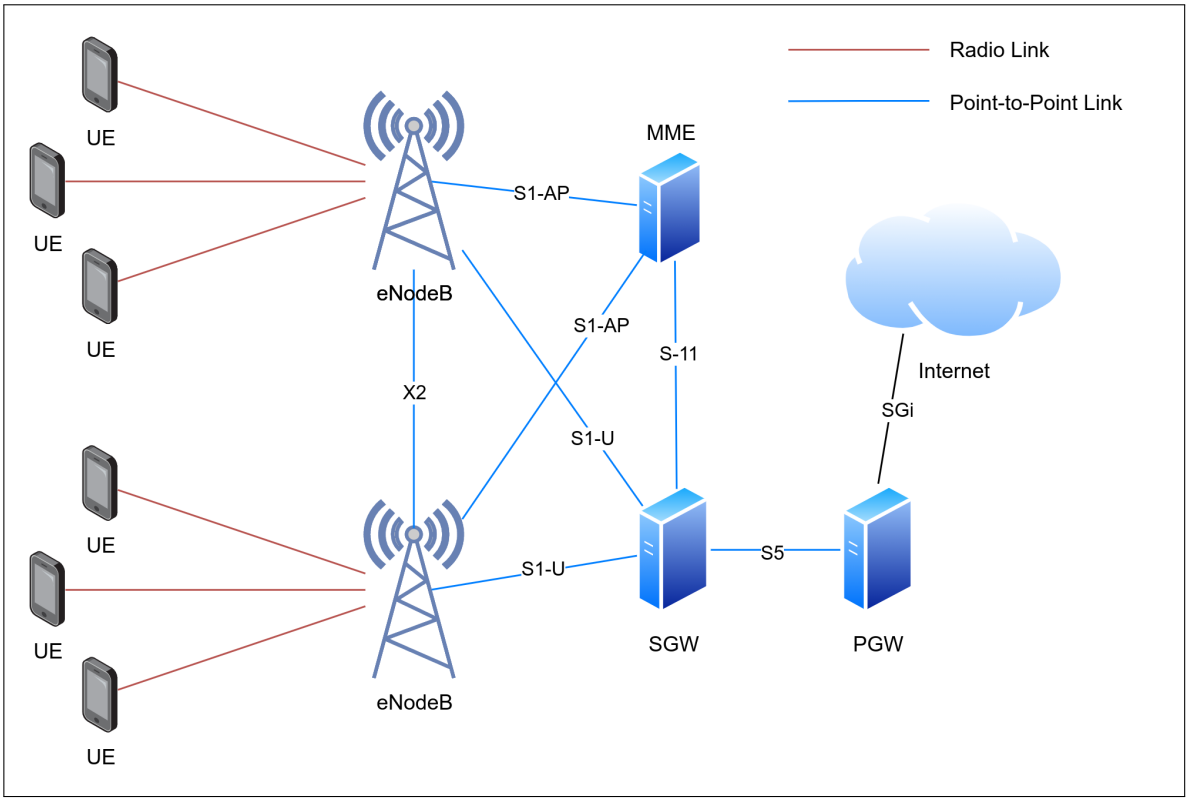

Figure 4.2: LTE-EPC NS-3 Module Overview

All UE nodes are interconnected by the LTE model to a specific eNodeB over the radio link, as indicated by the red edges. Moving up in the LTE hierarchy, all further system components (including the eNodeB) belong to the EPC model and their corresponding interfaces are simulated using Point-to-Point Channels: The X2 interface interconnects eNodeB nodes and is used to carry out UE handover procedures. The MME (Mobility Management Entity) node is responsible for assigning each UE to one Serving Gateway (SGW) per session; Thereby, the MME is communicating to eNodeB nodes over the S1-AP interface, while control messages between the MME node and the SGW are exchanged over the S-11 interface. Most relevant for the last-mile LTE simulation in this project are the user plane interfaces S1-U (interconnecting eNodeB nodes with SGW nodes) as well as S5 (interconnecting SGW with Packet Gateway (PGW)) nodes. Both interfaces S1-U as well as S5 realistically model the encapsulation of data packets using the GTP (GPRS Tunneling Protocol), UDP, *resp.*, IP protocols, as performed in real-world EPC [5]. Finally, the PGW node incorporates the role of a gateway between the LTE-EPC domain and other PDN (Packet Data Networks), *e.g.*, the internet. The communication among PGW and foreign PDN is carried out over the SGi interface [5].

It should be mentioned that the above diagram could be interpreted as a hierarchy read from left to right, in the sense that the deeper nodes are in the core network (*i.e.* the right-hand side), they are responsible for routing more and more aggregated traffic from, *resp.*, to the nodes which are hierarchically lower *w.r.t.* the structure of LTE: For example, if in the depicted architecture, all UEs are accessing the internet simultaneously, the single PGW must route all of the traffic from the entire mobile network towards the internet, and vice-versa.

Initially, it was planned to leverage the hierarchical structure modeled by the LTE-EPC module, by providing a simulation scenario consisting of several independent last-mile LTE domains, consisting of several UEs communicating with one eNodeB, as well as one SGW and one PGW per zone: The idea behind that was that all UE nodes shall act as data consuming NDN nodes, whereas their (aggregated) traffic would have been routed towards their corresponding PGW, while the PGW would have acted as a domain (*resp.*, zone) supernode acting as a gateway between intra- and inter-domain level communication.

At an early stage of this project, a working LTE-EPC simulation scenario was developed, which allows to define several independent LTE last-mile networks (*resp.*, zones): However, it turned out to be impossible to install ndnSIM in the LTE-EPC context, as the LTE-EPC module is exclusively compatible with IP-based protocols, while NDN (as mentioned earlier) is an entirely different network layer protocol. An email conversation with Dr. Spyridon Mastorakis – former maintainer of the ndnSIM repository – revealed that ndnSIM 1.0 [13] features IP-based NDN interfaces, which allow to run ndnSIM as an overlay over the TCP, or, UDP protocol: In theory, this would make it possible to integrate ndnSIM in a simulated LTE-EPC domain. However, for two reasons the decision was taken to not simulate IP-based NDN: Firstly, IP-based NDN interfaces are only available in ndnSIM >1.0 and would need to be ported to ndnSIM >2.0, which is a potentially complex and highly time consuming endeavour. Secondly, the aim is to simulate highly performant last-mile NDN zones; However, running NDN as an overlay over TCP or UDP would introduce additional protocol overhead, compared to operating NDN natively.

For these reasons, it was decided that last-mile zones shall be modeled in terms of a custom (ndnSIM-compatible) simulation topology, which fulfils the following two criteria:

- Each last-mile zone consists of several data consuming mobile nodes, which are associated with a hierarchically higher zone supernode, to effectively simulate the real-world hierarchical structure of LTE and EPC.
- All last-mile mobile nodes communicate over a shared-bandwidth channel, which realistically mimics the performance of the LTE radio link.

To achieve the latter, it was necessary to conduct a performance measurement experiment to derive realistic LTE performance measurements to be applied in the custom last-mile zone configuration. This was realized by re-using the aforementioned LTE-EPC simulation program, as described in the upcoming section.

4.2.2 LENA Performance Measurements

To collect realistic LTE performance metrics measured within last-mile networks, NS-3 script `lena-performance-measurements.cc` was developed, of which the source code can be found in section A.1. This script is a slightly modified version of the earlier mentioned simulation which initially was planned to be used to deploy ndnSIM in several autonomous LTE networks.

LTE-EPC Network Setup

Theoretically, this script is able to simulate a scenario involving several LTE networks by applying only very little modifications, however, for measuring LTE performance, we merely focus on one network as well as 5 UE nodes, as indicated on lines 43 and 44.

From lines 65 – 83 the LTE as well as EPC models are set up, using helper classes `LteHelper` as well as `PointToPointEpcHelper`: This involves setting up the IPv4 net masks required for the EPC interfaces, which are simulated in terms of Point-to-Point Channels [5] (*cf.* Figure 4.2). Furthermore, the EPC and LTE models are associated with each other using a method on the `LteHelper` object (l. 83). Thereby, both the LTE as well as EPC models are used in the default configuration provided by their corresponding NS-3 implementation [5].

Next, from l. 92 – 115 the Node objects corresponding to one single eNodeB as well as 5 UEs are instantiated and the `ns3::ConstantPositionMobilityModel` – a component of the NS-3 Mobility module – is configured and installed on them: The NS-3 Mobility module allows for simulating the placement as well as movement of Nodes in the context of a metric, three-dimensional Cartesian coordinate system [5]. Using the Mobility module is especially beneficial in simulations modelling over-the-air links involving propagation delay *resp.* propagation loss models, such that node connectivity decreases, the higher the simulated distance among two communicating nodes. In this scenario, we use constant

node positioning (*i.e.*, no movement, hence the name `ConstantPositionMobilityModel`): The eNodeB node is placed at the point of origin `(0, 0, 0)`, while the UE nodes reside at a distance according to multiples of 500 m towards the eNodeB: `(500 * n, 0, 0)`, `n` is the UE node index ranging from 0 to 4.

On lines 117 – 139, the installation of LTE-model specific Net Devices on UE and eNodeB nodes takes place, followed by setting up the IPv4 protocol stack on UE nodes and configuring routing to enable connectivity among UE and EPC nodes.

Finally, all 5 UE nodes are attached to the single eNodeB (lines 142 – 145). This completes the configuration of a functioning LTE-EPC network simulation, providing the possibility to install internet-specific Applications in the given scenario.

Performance Measurement Configuration

To measure the performance of LTE in terms of data rate as well as transmission delays, we install the NS-3 `UdpClientServer` applications [7] on the eNodeB as well as on the 5 UE nodes: The `UdpClientServer` application group can be used to simulate a UDP datagram flow from one (or more) clients to one UDP server. In our LTE simulation, we install a `UdpServer` on the single eNodeB and program the application to run between seconds 1.0 and 10.0 in simulated time (lines 149 – 156). On each of the 5 UE nodes, a `UdpClient` application is installed and configured to transmit each 0.05 s one UDP datagram of size 1500 B to the server on the eNodeB. This payload size corresponds to the NDN packet size eventually simulated. In comparison to the `UdpServer` application, the `UdpClient` applications are programmed to start their operation one (simulation) second later, to ensure the server is fully set up and ready to receive data, as soon as the first datagrams arrive from the clients (lines 159 – 173).

The illustration below depicts the LENA network scenario used to take performance measurements:

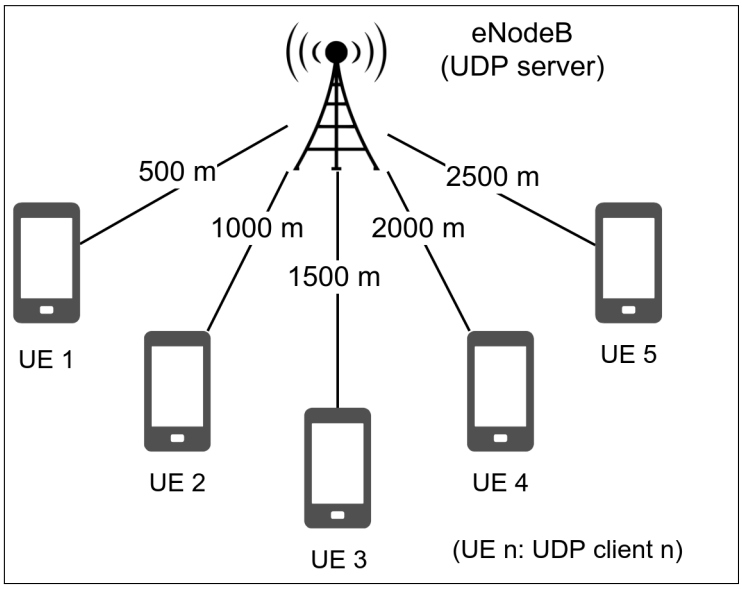

Figure 4.3: LENA Performance Measurement Scenario

The simulation script is concluded with setting up `FlowMonitor`, an NS-3 tool allowing for measuring the performance of IP-based packet flows between any two Nodes (lines 176 – 178) [5]. `FlowMonitor` is installed on all of the Nodes in the simulation, thus, the five UEs and the eNodeB in question are covered. After the NS-3 calls to run the scripted simulation and stop it after second 12.0 in simulation time (lines 180 – 181), the collected `FlowMonitor` statistics are exported to a file in XML format.

Measurement Results

The exported `FlowMonitor` statistics file holds five packet flow summaries according to the five UEs (running `UdpClient` applications) streaming UDP datagrams to the `UdpServer` application at the eNodeB. Each flow summary features several different statistics, *e.g.*, the number of transmitted packets. To determine the individual average transmission delay per packet (of size 1500 B), we divide for each flow the total measured transmission delay (`delaySum`) by the number of transmitted packets (`rxPackets` = 160 packets):

Table 4.1: Measured Individual LTE Transmission Delays

	Distance to eNodeB	`delaySum`	**Avg. Packet Transmission Delay**
UE 1	500 m	2'389'967'680 ns	14.937 ms
UE 2	1'000 m	2'390'168'480 ns	14.939 ms
UE 3	1'500 m	2'549'967'680 ns	15.937 ms
UE 4	2'000 m	2'550'168'480 ns	15.939 ms
UE 5	2'500 m	2'550'369'280 ns	15.940 ms

Furthermore, we repeat the performance measurements to determine the total throughput achieved by the five simultaneously streaming UEs. This time – however – all UEs are positioned at the same distance of 100 m towards the eNodeB to measure equal transmission conditions.

Individual throughputs are derived by dividing for each flow the total number of received bits (`rxBits`) by the transmission delay `delaySum`. This is presented in the following table, while the last row indicates the summarized total throughput achieved by 5 UEs. There were no packets dropped during the simulation, thus, for all flows `rxBits` corresponds to 160 transmitted packets, *i.e.*, 1'955'840 b (including protocol overhead).

Table 4.2: Measured Individual and Total LTE Throughputs

	`delaySum`	**Measured Throughput**
UE 1	2'389'967'680 ns	818.35 kbps
UE 2	2'390'168'480 ns	818.29 kbps
UE 3	2'549'967'680 ns	818.22 kbps
UE 4	2'390'570'080 ns	818.12 kbps
UE 5	2'390'770'880 ns	818.08 kbps
		4091.08 kbps

4.2.3 Custom Last-Mile Network Configuration

To replace the ndnSIM-incompatible LTE-EPC last-mile simulation, we define a custom last-mile network design, which merely mimics LTE, by applying the performance measurements taken as described in the previous section. This custom setup is plainly based on NS-3 `PointToPoint` Channels [5], which allow for connecting exactly two Nodes:

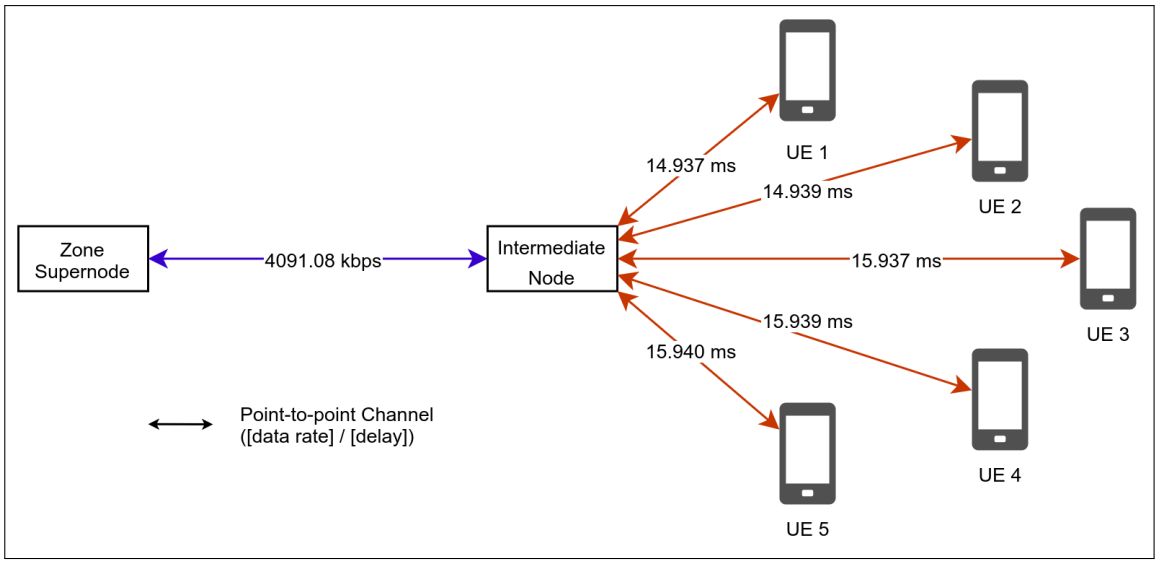

Figure 4.4: Custom Last-Mile Network Design

It should be noted that the depicted topology corresponds to one last-mile zone of the proposed architecture as displayed in Figure 4.1, however, the Caching Node is denoted "Intermediate Node" in this instance.

Each of the `PointToPoint` Channels is configured in terms of data rate as well as propagation delay: The individual channels between each UE and the Intermediate Node are parametrized with the measured transmission delays (*cf.* Table 4.1).

To simulate shared bandwidth among 5 UEs, we define a single `PointToPoint` Channel between the Intermediate Node and the Zone Supernode, on which the measured total throughput of 5 UEs is applied (*cf.* Table 4.2).

At this stage, the data rate on the `PointToPoint` link on the intermediate link (in purple, between the Intermediate Node and the Zone Supernode), as well as the propagation delays on the individual links among UEs and the Intermediate Node (in orange) have been specified. To prevent NS-3 from applying arbitrary default values *w.r.t.* propagation delay on the intermediate link, *resp.* data rate on the individual UE links, the default Channel settings are intentionally overwritten: On the intermediate link an insignificantly small propagation delay of 75 ns is set, while each of the five UE links is configured with a comparatively high data rate of 1 Gbps.

With this configuration, we effectively model propagation delay bottlenecks in terms of the individually applied delays on the UE links, while the single link between Intermediate

Node and Zone Supernode materializes a shared bandwidth channel by simulating a data rate bottleneck behind the five UE links. Thereby, this custom last-mile network design maintains the hierarchical structure of LTE, as all the traffic from, *resp.*, towards the UEs has to be transported by the single Zone Supernode.

4.3 Bridging of ICN and DHT Technologies

A crucial element for realizing the layered architecture is the bridging of ICN with DHT technologies, to enable data exchange between the intra- as well as the inter-domain communication layer. This section covers how this was accomplished in the context of the chosen simulation tools, *i.e.*, ndnSIM as well as `ns-3-chord`.

In the proposed architecture (*cf.* Figure 4.1), mobile nodes are considered as data consumers, which – in the ndnSIM context – send out NDN interest packets, to retrieve data from upstream nodes. On the other hand, the zone supernodes are assigned the role of data producers, which are not only responsible for storing all items of the simulated data universe, but also for translating incoming interests from the NDN domains into retrievals on the Chord overlay, and eventually responding to the NDN downstream upon successful reception. With that being said, the zone supernodes can be described as the only gateways part of the suggested architecture, which act as a passthrough from NDN to Chord.

To equip zone supernodes with the ability to bridge between ndnSiIM and `ns-3-chord`, `ChordProducer` was developed – a custom ndnSIM application, as described in the upcoming section.

4.3.1 ChordProducer ndnSIM Application

The source code of the `ChordProducer` ndnSIM application is attached to section A.2 and consists of C++ header file `ndn-chord-producer.hpp` as well as the according implementation file `ndn-chord-producer.cpp`. The line references in this section – if not noted otherwise – relate to the implementation file in subsection A.2.2.

`ChordProducer` is based on the default ndnSIM application `Producer` [27, 28], which processes new incoming NDN interest packets by responding with a data packet holding virtual payload. Thereby, the standard `Producer` app has an attribute called `Prefix`, with which the NDN interest filter can be configured. Let us consider the following example to illustrate that: A node running a `Producer` application configured on prefix `/data` receives an interest packet for data item `/data/7` (7 is this item's sequence number); Since the prefix matches, the `Producer` application gets triggered and immediately responds with a data packet of virtual payload back to the downstream, such that the interest is satisfied [27, 28].

However, instead of instantly issuing data back to the downstream, `ChordProducer` – in contrast to `Producer` – triggers a data object retrieval on the DHT, by querying the sequence number of the NDN interest from the Chord ring. For `ChordProducer` to be able to interact with `ns-3-chord`, a new attribute was added to the class (as seen on lines 51 – 55), with which one can pass a pointer to the `ChordIpv4` installed on the supernode.

In general, for the `ChordProducer` passthrough to function properly, the following requirements must be fulfilled:

- A pointer to the supernode's `ChordIpv4` instance is passed to its `ChordProducer` application, providing `ChordProducer` with access to the Chord `Retrieve` routine.
- The Chord DHT has been completely set up, *i.e.*, all participating supernodes and data objects present in the simulated data universe have been inserted into the DHT. Thereby, the keys of the objects stored in the DHT must correspond to the simulated NDN data item sequence numbers.
- The `ChordProducer` application is configured to serve data on the same name prefix as the prefix specified in incoming interest packets from the local mobile nodes: This is to ensure that all new interests incoming from mobile consumer nodes are processed by the `ChordProducer` application on the local last-mile zone supernode.

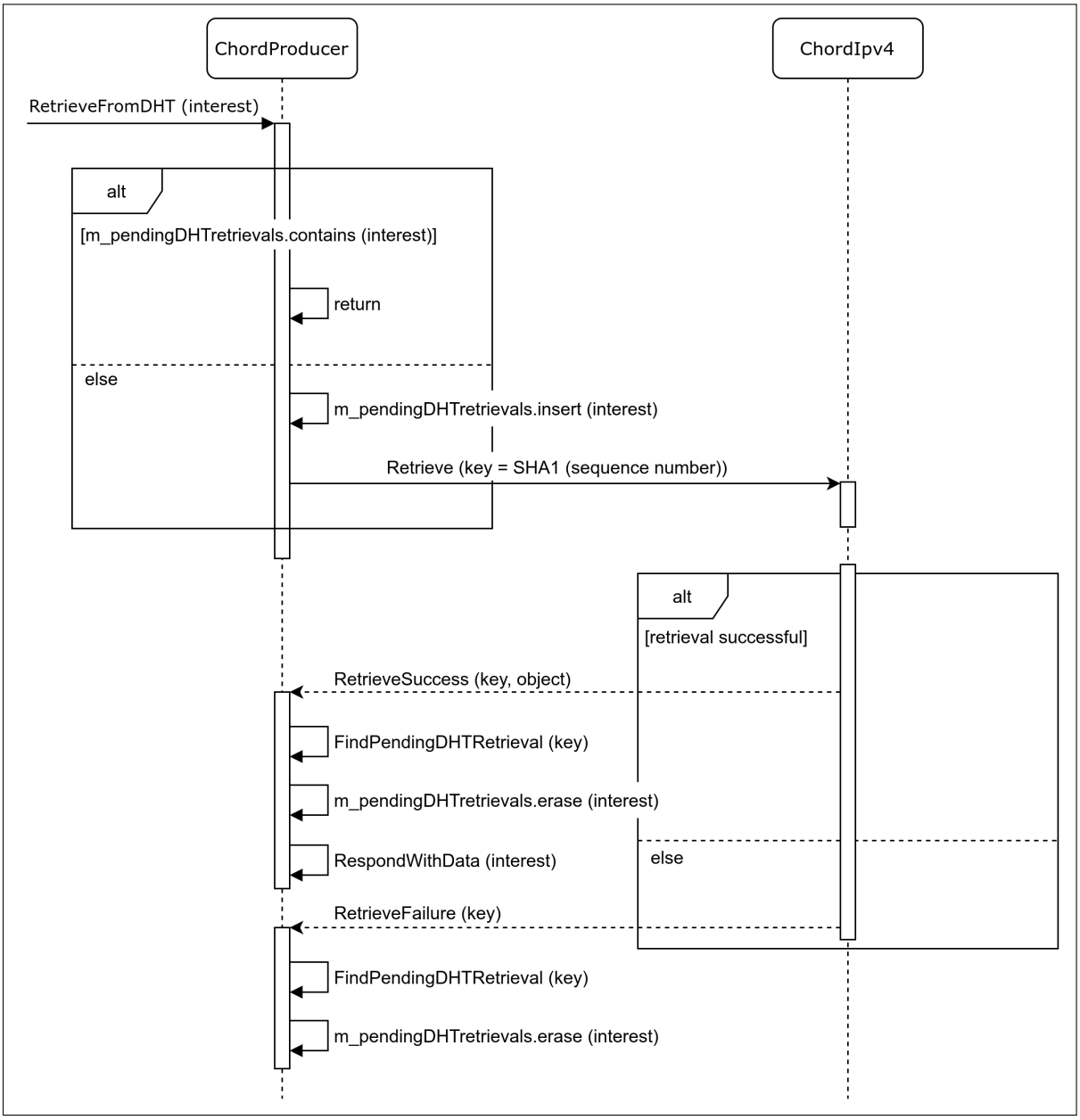

Figure 4.5: ChordProducer Application Sequence Diagram

The above sequence diagram illustrates the mechanism triggered by `ChordProducer` to translate an NDN interest into a DHT retrieval, and eventually responding with data towards the downstream:

The entry point is method `RetrieveFromDHT` (*cf.* lines 132 – 147), which automatically gets invoked, as soon as an interest packet arrives at the supernode, for which the data is not (yet) cached in the Content Store. In scenarios in which NDN interest packets arrive at a high pace at the supernodes, it can happen that identical interests arrive almost simultaneously. For this case, `ChordProducer` keeps a C++ STL `unordered_set` denoted `m_pendingDHTretrievals` (definition on line 78 in the header file), which is used to record ongoing Chord retrievals: If an incoming interest is already contained in this set, the method immediately returns, to prevent the Chord overlay from being unnecessarily overloaded with identical retrievals from the same supernode.

In the opposite case, the DHT has not yet been queried for the data object in question, thus, the new interest object is inserted in `m_pendingDHTretrievals`. To retrieve the corresponding data object from the DHT, the method extracts the sequence number from the interest object and hashes it using SHA1: The resulting hash is used as the `key` parameter to invoke the `Retrieve` routine on the associated `ChordIpv4` application, to which `ChordProducer` points to.

At this stage, handling the lookup and the transmission of data is handled by the `ns-3-chord` DHT layer. Such that `ChordProducer` is eventually notified of the successful, *resp.*, unsuccessful DHT retrieval, it subscribes to the retrieve success as well as retrieve failure callbacks on the `ChordIpv4` application (*cf.* lines 70, 71).

If the Chord layer calls back some data object retrieval as successful, the `RetrieveSuccess` method (*cf.* lines 187 – 199) gets invoked while the hashed key is passed as one of the parameters. The next task is for `ChordProducer` to determine to which interest object the retrieved object belongs to: This is implemented in method `FindPendingDHTRetrieval` (*cf.* lines 163 – 185), which iterates through the `m_pendingDHTretrievals` set, and – using SHA1 – hashes the sequence number of every interest object in the set; During each iteration, the digested sequence number is compared with the key reported back by `ChordIpv4` (*cf.* line 174). If the key and the digest match, the current interest must be the one of which its data object was retrieved successfully from the Chord DHT. Thus, the interest object can be erased from the set of pending interests (*cf.* line 196). Finally, method `RespondWithData` (*cf.* lines 98 – 130) is invoked, which causes the supernode to respond with an NDN data packet to the downstream nodes.

In case the DHT retrieval was unsuccessful (*e.g.*, due to failed data object transmission), `ChordIpv4` calls back a retrieve failure, which gets received in terms of the object key in method `RetrieveFailure` (*cf.* lines 201 – 212). Also in this unsuccessful case, the corresponding interest object is searched and removed from `m_pendingDHTretrievals`, such that for future occasions, it would be possible to retry retrieving some data object from the Chord layer. After that – however – the method terminates, as due to the failed DHT retrieval the interest is not being answered with any data packet.

4.4 Data Object Popularity Simulation

This section describes the modeling of object popularity, which refers to the frequency pattern at which resources (*resp.*, pieces of information) are requested by network participants. In the simulated architecture, network participants are represented by nodes, while network resources exist in the form of data items. Thus, the goal was to set up mobile data consumer nodes to request data items according to a model representing real-world demand for data resources.

As for a foundation for an object popularity model, the decision was taken to choose Zipf's law, which is introduced in the following section. The subsequent subsection 4.4.2 presents how Zipf's law was integrated in the intra-domain layer simulation.

4.4.1 Zipf's Law and the Internet

Zipf's law is a principle first described by American linguist G.K. Zipf: When studying a large collection of English texts, Zipf detected and described the phenomenon, that while a very small set of words account for the highest occurrence frequency, a large set of remaining words only appear very seldom [40]. This becomes immediately clear when considering an example: In arguably most texts, articles such as 'the' occur by far most frequently, while the opposite is true for a large proportion of infrequent words, *e.g.*, words of foreign origin only used in specific contexts. If considering word occurrence in terms of a word frequency rank table (*i.e.*, the first rank is held by the most frequently occurring word), Zipf's law can be formulated as such that in large texts, words' rank in the frequency table is inversely proportional to their frequency of occurrence [17].

Zipf's law not only applies to word occurrences in texts, but it can also be observed in a vast number of different fields and contexts. For example, already in the early years of the World Wide Web, it was shown that several aspects of the internet (and its usage) follow Zipf's law: In a 2001 article, Adamic and Huberman present the following four different correlations involving websites and demonstrate that they correlate with Zipf's law [12]:

- **Proportion of sites** *vs.* **number of pages**: Most websites consist of merely one page, while only a very small amount of sites contains a vast number of pages.
- **Proportion of sites** *vs.* **number of users**: Only a small proportion of all sites has millions of unique, active users, while the greater part of websites is only visited very seldomly.
- **Proportion of sites** *vs.* **number of out-links**: Most websites only hold a handful of out-links (towards other websites), while the amount of sites pointing to thousands of other sites is very small.
- **Proportion of sites** *vs.* **number of in-links**: A minority of popular websites has thousands of in-links (from other websites), while a majority of sites is only pointed to by a handful of inbound links.

In the scope of this work, we focus on the second correlation above (*i.e.*, proportion of sites *vs.* number of users): An example to that could be popular news websites, which get visited by a disproportionately large share of users, compared to static, informational websites of small businesses. Although this example involves websites and users, the correlation can also be reduced to the dimensions data items and data requesting nodes, as technically, website users operate some type of internet-ready device (*resp.*, node) running a web browser, to retrieve news articles, which consist of data items served from the web server node hosting some news website.

With that being said, the Zipfian correlation among website users and websites can be described as compatible with the proposed architecture: The goal is therefore to put into place a data retrieval application on the mobile consumer nodes which causes them to request a small amount of data items at a disproportionately high frequency, while the remaining items of the simulated data universe are requested only very seldomly.

About Zipf's law, it should be mentioned that its mathematical representation law belongs to the class of power law distributions. As such, Zipf's law can be expressed as follows: The probability that some word in a large text appears at an occurrence frequency x corresponds to x^{-s}, where ≥ 1. In general, all power law distributions (including Zipf's law) are scale-free, meaning that independently of the size of the distribution, the underlying characteristics remain the same [12]. For simulating object popularity, this implies that even if it would be unmanageable to simulate the entire data universe present in the internet, real-world data request patterns can still be simulated analogously at the small-scale.

4.4.2 Simulating Data Object Popularity on the Intra-Domain Level

To simulate data object popularity according to Zipf's law, the `ConsumerZipfMandelbrot` application [25] is installed on all data consuming mobile nodes. `ConsumerZipfMandelbrot` is part of the ndnSIM reference applications and implements the Zipf-Mandelbrot distribution, a generalization of the basic mathematical representation of Zipf's law [26]. The `ConsumerZipfMandelbrot` application ensures to send out interests for low sequence numbers (*i.e.*, popular items) more frequently than for high sequence numbers (*i.e.*, unpopular items). This is realized in the following way:

During initialization, the `ConsumerZipfMandelbrot` considers the specified total number of data items (*resp.*, sequence numbers) and using the Zipf-Mandelbrot function, produces a vector of cumulative probabilities `Pcum`, which later on is used to draw sequence numbers to issue interests for. The `Pcum` vector is produced as follows [25]:

1. `Pcum` is initialized as a vector of size $N+1$ holding floating point numbers of double precision. N is the number of unique data item sequence numbers in the simulated data universe. The first element, `Pcum[0]` is assigned value 0.0.

2. By iterating through all sequence numbers `i` (ranging from 1 to N), the cumulative probability `Pcum[i]` for each sequence number `i` is computed according to:
   `Pcum[i] = Pcum[i - 1] + 1.0 / std::pow(i + q, s)`

Thereby, the expression after the `+` operator corresponds to the Zipf-Mandelbrot probability function, *i.e.*, $P(i) = 1/(i+q)^s$ [26]

3. Finally, the `Pcum` vector is normalized, by dividing each cumulative probability by the cumulative probability of the last element, *i.e.*, `Pcum[N]`:
   `Pcum[i] = Pcum[i] / Pcum[N]`
   This ensures that all cumulative probabilities occur within the interval $(0.0, 1.0]$.

With a number of data items $N = 10$, a parameter of power `s` $= 1.2$ and `q` $= 0.7$, `ConsumerZipfMandelbrot` produces the following vector of cumulative probabilities, `Pcum`:

Table 4.3: Exemplary Cumulative Probability Vector

Cumulative probability [1] = 0.30986
Cumulative probability [2] = 0.487715
Cumulative probability [3] = 0.609575
Cumulative probability [4] = 0.701026
Cumulative probability [5] = 0.773578
Cumulative probability [6] = 0.833339
Cumulative probability [7] = 0.883911
Cumulative probability [8] = 0.927591
Cumulative probability [9] = 0.965924
Cumulative probability [10] = 1.0

To determine the next data item (*i.e.*, a sequence number) for which to issue an interest packet for, `ConsumerZipfMandelbrot` draws a random number from a uniform distribution within the interval (0.0, 1.0], by using the NS-3 built in random number generator [4]. By iterating through `Pcum`, the drawn random number is compared to the individual cumulative probabilities of the data items: Eventually, the data item with the smallest cumulative probability, which is larger than, or, equally great as the random number, gets chosen to issue an interest packet for. For example, if random number 0.58 gets drawn, in the above example the chosen sequence number would be 3, as its corresponding cumulative probability is the smallest greater than the random number ($0.61 >= 0.58$) [25]. Thus, `ConsumerZipfMandelbrot` would let the consumer node issue an interest packet for `/[prefix]/3`.

As mentioned, the random number is drawn from a uniform distribution, *i.e.*, there is a uniform probability for any number between 0.0 and 1.0 to be chosen randomly. Compared to the uniform distribution of random numbers, the cumulative probability distribution is skew, as for in between lower sequence numbers, the probability grows disproportionately faster compared to among the upper sequence numbers. This implies that the random drawing mechanism implemented in `ConsumerZipfMandelbrot` will have the tendency to draw low sequence numbers (*i.e.*, data items which are considered popular) more frequently compared to the upper end of the simulated data universe: Thus, the data item request frequency distribution as well as the consequent data item popularity pattern fulfil Zipf's law.

4.5 Inter-Domain Level Communication

Concerning the inter-domain layer of the proposed architecture, the DHT overlay based communication among different zones has already been discussed in section 4.3. The missing element for a complete working simulation scenario is the definition of an IP-based transit network among the supernodes, which serves as an underlay for inter-domain communication over Chord.

In the internet, the interconnection among foreign domains (*e.g.*, networks operated by individual commercial Internet Service Providers) is realized through internet backbone networks: Internet backbones can be characterized as transit networks consisting of high bandwidth, long distance links which provide connectivity among remote domains as well as high performance core routers [22]. To realistically model inter-domain level communication, it was decided to model a transit network which reflects the characteristics and structure of a backbone network. To accomplish that, the following contribution was taken as a reference:

In a 2015 paper [29], S. Nikolaev et al. from the Lawrence Livermore National Laboratory present the findings of large NS-3 simulations which involved up to ten billion simulated nodes in planetary-scale topologies: Their simulation is based on a highly distributed implementation, *i.e.*, large simulated network topologies are partitioned into up to a thousand individual parts, while all subdivisions are executed in parallel on a high performance computing cluster. This is realized by the NS-3 built-in parallel scheduler module, which supports the Message Passing Interface (MPI) used in parallel computing. In the course of simulating large-scale networks in parallel, S. Nikoalaev et al. compare the performance of the default NS-3 parallel scheduler against a custom designed solution. To model a transit network among router nodes, they produced topologies in terms of small-world networks using the Watts-Strogatz random graph generation algorithm, to effectively mimic the characteristics of a real-world backbone network [29].

To simulate an inter-domain transit network approximating the topology of internet backbones, the random small-world network generation approach suggested by S. Nikolaev et al. [29] was reproduced. The following section introduces small-world network graphs as well as the Watts-Strogatz random graph generation algorithm.

4.5.1 Small-World Networks and the Watts-Strogatz Model

Small-world graphs *resp.* networks incorporate the commonly known small-world phenomenon, which describes that there is a connection between any two strangers over a small concatenation of acquaintances, *i.e.*, each person is at the maximum only six people 'away' from any other person on the planet [36]. In the past, several empirical studies have shown that a variety of systems have small-world properties, such as the internet, social networks [36], or the electricity grid in the western USA [37]. In small-world graphs, this phenomenon translates to the property that on average, in a network of size N, the distance (*resp.*, the number of hops) required to reach any node from another corresponds to $\log(N)$ [37].

The Watts-Strogatz model introduced in 1998 describes an algorithm which can be used to randomly generate small-world networks: As a starting point, the algorithm defines a lattice, *i.e.*, a meshed graph, in which each vertex is connected to its k nearest neighbour vertices using undirected edges. The algorithm then iterates through all of the edges and at at a probability p rewires the edge randomly, *i.e.*, an edge will not connect some vertex A with one of its k nearest neighbours anymore, but connects A with a randomly chosen different vertex from anywhere else in the graph [37].

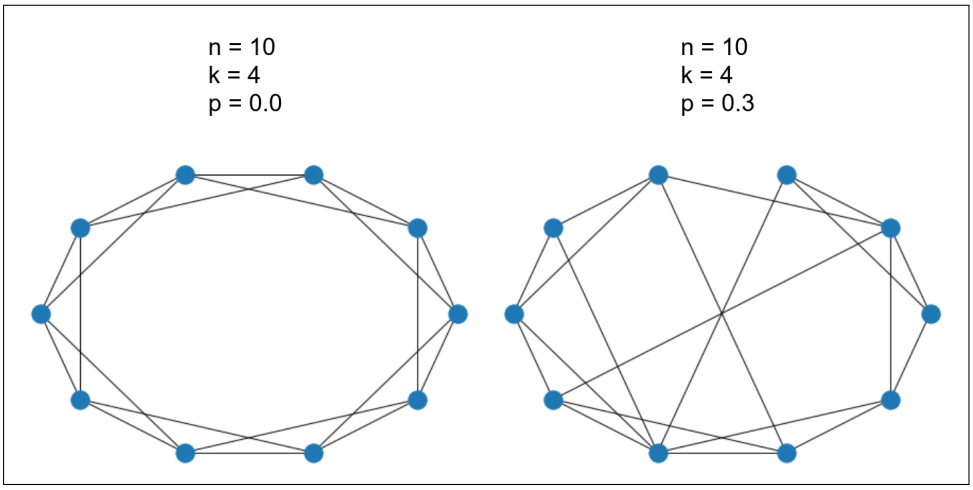

Figure 4.6: Exemplary Watts-Storgatz graphs

The above figure depicts two exemplary graphs generated using the Watts-Strogatz algorithm. Both graphs consist of $n = 10$ vertices, while each of the vertices is connected to its k nearest neighbours. The rewiring probability for the left-hand side graph was $p = 0.0$, thus, none of the edges were randomly rewired and the graph corresponds to a lattice showing no small-world properties whatsoever. The right-hand side graph – however – was generated with a rewiring probability $p = 0.3$, thus, some of the edges connect non-neighbouring vertices across the graph.

The higher the chosen rewiring probability p, the stronger is the small-world effect in the generated graph, *i.e.*, the more edges shortcut remote regions in the graph. In general, Watts-Strogatz small-world graphs generated with intermediate p values exhibit two main properties, *i.e.*, high clustering as well as small-world properties (already discussed): High clustering refers to the fact that per default, vertices are connected to their neighbouring vertices, which leads to multiple clusters of interconnected vertices [37].

It can be argued that backbone networks embody the same two properties, as router nodes tend to have a high degree of connectivity to neighbour nodes in their proximity, which translates to the clustering property in graph theory. At the same time, small-world properties can be identified at backbone networks' domain-spanning long-distance links which ensure connectivity among remote nodes over a small number of required hops.

4.5.2 Small-World Random Network Generation

To generate small-world graphs that determine the topology of simulated inter-domain transit networks, the `watts_strogatz_graph` random graph generation function part of NetworkX was used: NetworkX is a prominent Python library used in the field of network analysis [24].

Running the following Python code generates, draws and exports a Watts-Strogatz small-world network according to a network size of `n` $= 8$, with edges to `k` $= 3$ nearest neighbours for each vertex and a rewiring probability of `p` $= 0.7$:

```
1 import networkx
2 G = networkx.watts_strogatz_graph (n = 8, k = 3, p = 0.7)
3 networkx.draw_circular(G)
4 networkx.write_edgelist(G, "WattsStrogatz.edgelist", delimiter=",")
```

The random graph is created with the call of function `watts_strogatz_graph` on the second line. On the subsequent line, the graph is drawn in a circular shape, to produce a representation as seen in the examples in Figure 4.6. Finally, with function `write_edgelist`, all the edges of the generated graph are exported in terms of edge coordinates to a comma-separated file: If the graph contains *e.g.*, an edge between vertices 1 and 6, the file will include a line indicating edge coordinates `1,6`. The exported edge list is eventually imported in the simulation scripts to define the transit network. Thereby, for each line (*resp.*, for each edge), a `PointToPoint` Channel among the Nodes with indices according to the edge coordinates is defined: To simulate high-performance backbone links, the data rate is set to 100 Mbps and the propagation delay to 10 ms.

It should be mentioned, that due to the involved randomness, the Watts-Strogatz algorithm might generate unconnected graphs, *i.e.*, graphs in which – per the lack of a connecting edge – certain areas are isolated from the rest of the graph. For the planned architecture, a connected transit network is a necessity, such that the connectivity among any two domains can be guaranteed. To generate connected Watts-Strogatz graphs, NetworkX provides function `connected_watts_strogatz_graph`: Compared to `watts_strogatz_graph` this variant iteratively generates a new graph, and after each try it checks, whether the new graph is connected. If during ≤ 100 tries, a connected graph is derived, it gets returned, otherwise, an error messages is thrown [24].

4.6 Plain NDN Reference Architecture

This section describes the plainly NDN-based architecture, which was developed as a reference for to compare the proposed layered NDN & Chord architecture (*cf.* Figure 4.1) against. Topology-wise, the plain NDN architecture is identical to NDN & Chord, as it – too – defines several last-mile zones with a fixed number of mobile nodes, an intermediate caching node as well as a zone supernode each: Thereby, the inter-domain level communication among supernodes is transported over a small-world transit network as described in the previous section. Technology-wise – however – this plain NDN architecture does not define an IP-based inter-domain communication layer, but rather, NDN is used as the single primary communication technology; This allows for simulating scenarios in which NDN is deployed at the global scale.

The following illustration gives an overview of the plain NDN reference architecture:

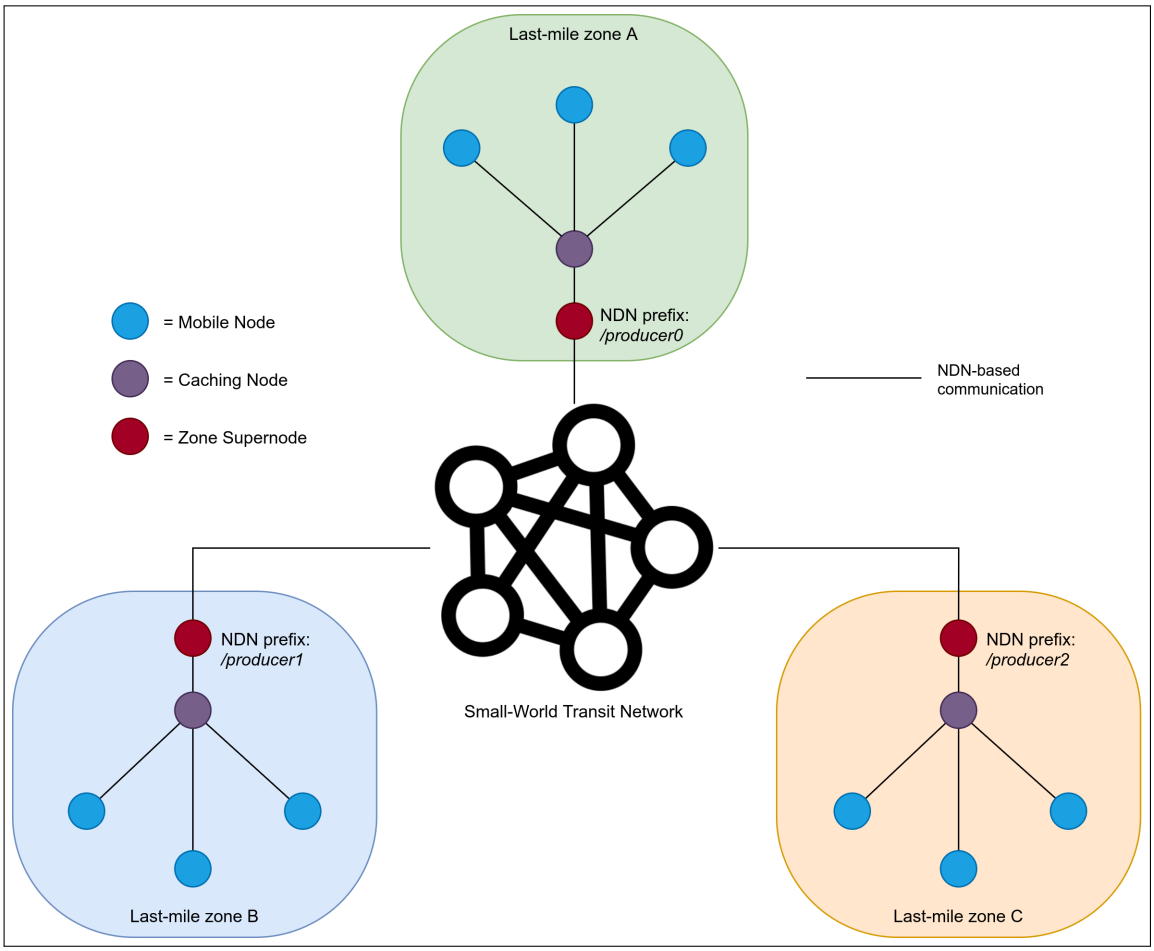

Figure 4.7: Plain NDN Reference Architecture

In the NDN & Chord architecture, inter-domain communication is entirely realized by Chord, which – with its SHA-1 hashing based namespace – provides load balancing among all data producing supernodes (*cf.* subsection 2.4.2): More specifically, given a data universe of a certain size, Chord ensures to distribute the responsibilities over data objects

consistently among the supernodes, such that every supernode is obliged to serve an approximately equal share of the total number of simulated data objects. To achieve a similar behaviour in this plain NDN reference architecture, a custom ndnSIM application setup was materialized, as described in the upcoming section.

4.6.1 ndnSIM Application Setup

Producer Application Configuration

Per default, scenarios simulating the plain NDN reference architecture require the deployment of the standard ndnSIM Producer application, which immediately initiates sending back data packets upon incoming interest packets with matching prefix (*cf.* section 4.1.2). Compared to NDN & Chord, the plain NDN architecture defines a different management of name prefixes, *i.e.*, each supernode's Producer application is configured to serve a unique prefix. This allows for installing domain-specific Producer applications on the supernodes, such that *e.g.*, the supernode of the second zone will respond to interests with prefix `/producer1`, while the fifth zone produces data for prefix `/producer4`.

Setting a unique prefix filter on each Producer application effectively assigns individually served namespaces to the zone supernodes. To achieve load balancing among the supernodes, the mobile consumer nodes in the last-mile zones are therefore required to construct interest packets with alternating prefixes, such that – depending on the specified prefix – interest packets are processed by different supernodes. To fulfil this requirement, a custom ndnSIM was developed, which is described in the following section.

Custom Consumer Application: ConsumerZipfBuckets

`ConsumerZipfBuckets` is a custom implementation based on `ConsumerZipfMandelbrot` (*cf.* subsection 4.4.2). The only difference to the `ConsumerZipfMandelbrot` application lies in the way interest prefixes are chosen. While all existing Consumer applications work with a static prefix, `ConsumerZipfBuckets` constructs interest packets by dynamically choosing alternating prefixes for each drawn interest sequence number: This is achieved by introducing the concept of prefix buckets, with which all of the simulated data items (*resp.* sequence numbers) are partitioned equally according the number of prefix buckets, which should be chosen according to the number of last-mile zones: For example, if `ConsumerZipfBuckets` is configured to work with 5 prefix buckets and the Zipf-Mandelbrot algorithm draws 17 as the upcoming sequence number, the consumer constructs an interest packet according to the first prefix bucket, *i.e.*, `/producer0/17`. In this setting, interests with sequence numbers 1 – 20 will thus be sent with prefix `/producer0`, while for sequence numbers 81 – 100 interests with prefix `/producer4` will be issued. Together with the appropriate prefix configuration of the `Producer` applications, this approach allows for a plain NDN scenario in which each Producer serves an equal share of the total amount of data items (*cf.* the indicated served prefixes in Figure 4.7), as it is the case with DHT namespace balancing in the NDN & Chord architecture.

4.6.2 Forwarding Strategies & Routing

As opposed to NDN & Chord, in the plain NDN architecture NDN traffic does exceed the last-mile zones and also traverses through the small-world transit network in terms of inter-domain communication. Due to the complex structure of the transit network, it seems reasonable to study the implications of using different NFD forwarding strategies leading to different routing behaviours among nodes part of the transit network.

It can be argued that the *Best Route* forwarding strategy is unsuitable to be deployed in large scale NDN networks: This stems from the fact that – being an information-centric technology – NDN (*resp.*, NFD) uses name prefix based FIB entries, compared to classical address-based IP FIBs; In general, in large networks the data universe (*i.e.*, the totality of all served data resources) is significantly larger than the host universe (*i.e.*, the totality of participating nodes), since most server nodes tend to provide multiple data resources. Globally deploying correct and complete FIB tables in large NDN networks is expected to be unattainable and thus, the *Multicast* strategy must be used as a remedy for *Best Route*.

In chapter 5, plain NDN simulation scenarios in which the *Multicast* forwarding strategy is deployed globally are studied: More specifically, worst-cases with extremely sparse FIB tables are considered, in which every node broadcasts (*resp.*, floods) all interest packets to all upstream neighbour nodes. The implementation details about how the *Multicast* strategy is configured to simulate interest *Flooding* are discussed in section 4.7.2.

4.7 Documentation of Simulation Scenarios

This section describes the two main NS-3 simulation programs (*resp.*, scripts) which were developed to evaluate the introduced ICN-DHT architecture. Their C++ source code can be found in the appendix under section A.3.

The first script, *i.e.*, `ndn-zones-chord-small-world.cc` (*cf.* subsection 4.7.1) simulates the proposed layered NDN & Chord architecture depicted in Figure 4.1. The second script, *i.e.*, `ndn-zones-buckets-small-world.cc` (*cf.* subsection 4.7.2) simulates the plain NDN reference architecture described in section 4.6.

Script `ndn-zones-buckets-small-world.cc` is to a great extent identical to `ndn-zones-chord-small-world.cc`. Therefore, the following description of the first script includes line references colour-coded in blue to the second one, whenever parts of the NDN & Chord script are identical to the plain NDN counterpart.

4.7.1 NDN & Chord Simulation Scenario

Chord Boilerplate Code

Simulation script `ndn-zones-chord-small-world.cc` (*cf.* subsection A.3.1) starts with `ns-3-chord` boilerplate code, *i.e.*, methods which are required to interact with the Chord layer. All functions are taken from the default `ns-3-chord` example simulation scripts, except for `ChordHelper::Insert` (l. 60 – 76): This function takes resource name and value strings as inputs and converts them to a key-value pair to be inserted in the Chord DHT: Thereby, the passed resource value string is appended with several ”v” characters (l. 68 – 72), until the value length corresponds to the specified payload size of 1500 B; This way, the simulated Chord object size is guaranteed to be the same as the packet size used in the NDN domains.

Main Simulation Variable Definition *

The actual simulation program is contained in the `main` function, starting at l. 134 (l. 37). On l. 137 – 141 (l. 40 – 44), the following main simulation variables (*resp.*, parameters) are defined:

1. `noUEs`: The no. of UE consumer nodes to be simulated in each last-mile zone
2. `noDataItems`: The no. of total data items in the simulated data universe
3. `interestsPerS`: The frequency in seconds according to which consumer applications issue interest packets
4. `simulationRuntimeM`: The amount of runtime in simulated minutes (excluding the initialization phase of the Chord layer)

5. `csSize`: The CS size in no. of packets to be used on nodes with activated caching

L. 159 – 187 (l. 62 – 84) define some further variables, constants and data structures, which will be referred to later on.

Set Up of Small-World Transit Network *

On l. 190 – 221 (l. 87 – 109) the transit network used for inter-domain communication is specified. First, a set of transit network nodes (denoted `smallWorldNodes`) is created (l. 190 (l. 87)); The default size of the transit network is 8 nodes. Such that Chord can operate over an IP-based underlay, the NS-3 `InternetStack` [5] is installed on all the transit nodes (l. 191, 192). Any two nodes of the transit network are interconnected by a `PointToPoint` link, which is configured on l. 194 – 196 (l. 89 – 91): The data rate is set to 100 Mbps and the propagation delay to 10 ms. Afterward, on l. 202 – 204 (l. 96 – 98) the transit network is constructed according to the edge coordinates contained in `smallWorldEdges`: This is a vector of integer pairs, which correspond to the edge coordinates exported by the NetworkX Watts-Strogatz graph generation function (*cf.* section 4.5): If the graph defines *e.g.* an edge among vertices 2 and 5, a `PointToPoint` link is installed which connects the second and fifth transit network nodes. To enable IP connectivity, `IPv4AddressHelper` [5] is used to automatically assign IPv4 addresses to all interconnected NetDevices (l. 207 – 209): As all NetDevices on the transit network nodes now have an assigned IP-address, it is possible to populate routing tables on all transit nodes (l. 213) using `Ipv4GlobalRoutingHelper` [5]. This makes all transit nodes into routers, such that the network manages to route Chord's IP traffic among any two source and destination transit nodes. Finally, a set of 5 randomly chosen transit nodes are defined as last-mile zone supernodes (l. 216 – 220, l. 104 – 108). The remaining transit network nodes are merely router nodes.

Chord Supernode Initialization

On the subsequent two code blocks (l. 224 – 264), the Chord layer is set up to enable inter-domain communication. By iterating through all last-mile zones, a `ChordIpv4` application is installed on each supernode (l. 232): The supernode of the first zone is defined as the Chord bootstrap node, thus, during each iteration its IP address is passed as a construction parameter for the `ChordIpv4Helper` object (l. 226). On all `ChordIpv4` objects, the necessary Chord layer callback functions are set, by referring to the `ns-3-chord` helper methods defined at the very beginning of the script. At this stage, the `ChordIpv4` application is ready to receive Chord instructions. By scheduling a call to the `InsertVNode` routine, one VNode is inserted to the Chord DHT for every supernode (l. 242 – 252), by iterating through all last-mile zones: During the first iteration, the bootstrap supernode is initialized for which a VNode with name "bootNode" is inserted (l. 244). For all further supernodes, the script inserts a VNode with a constructed name indicating the last-mile zone index, *e.g.*, "superNode1" (l. 250). Thereby, each VNode insertion is scheduled after an interval of `INTER_VNODE_INSERTION_GAP_S` = 70 seconds simulation time, such that

Chord has enough time available to (re-)organize keyspace responsibilities, as the supernode VNodes are joining the DHT ring. Finally, on l. 255 – 264 the simulated data items are inserted into the DHT. To balance the insertion load, the supernodes (*resp.*, their `ChordIpv4` applications) are cycled through using a modulo expression (l. 257). The key-value pair of the data item to be inserted is constructed as follows: The key is defined as a string holding the data item index, which corresponds to an NDN packet sequence number (l. 263). The value is defined as "payload" with a sequence number concatenated, *e.g.*, "payload57" (l. 258). Simulation-time-wise, the data item insertions are scheduled after the VNode initialization phase, *i.e.*, at `CHORD_INSERTION_STARTTIME_S` $= 5 * 70 = 350$ s (l. 179, 259); In-between item insertions, an interval of 0.5 s simulation time is used to ensure that all data item insertions will be successful.

Last-Mile Zone Configuration *

On l. 268 – 297 (l. 113 – 142), the individual last-mile zones are configured, according to the topology defined in subsection 4.2.3. Initially, the `PointToPoint` link between the zone supernode as well as the intermediate node is defined: The data rate is set to according to the measured achieved total rate for five LTE UEs (*cf.* Table 4.2) and a propagation delay of 75 ns is specified (l. 270, 273 (l. 115, 118)). In each last-mile zone, an intermediate node is created and the `PointToPoint` link is installed among intermediate and super node (l. 280, 281 (l. 125, 126)). Subsequently, for each zone the configured no. of last-mile UE nodes are created (l. 285 (l. 130)). To interconnect UEs with the zone's intermediate node, individual `PointToPoint` links are defined as follows:

- The propagation delay is chosen according to one of the 5 LTE UE measurements (*cf.* Table 4.1); If more than 5 UEs are defined per zone, the measured delays are re-used in a loop fashion (l. 292 (l. 137)).
- The link data rate is set to 1 Gbps (l. 294 (l. 139)).

Finally, each individual UE link is installed among the UE node and the corresponding zone's intermediate node (l. 295 (l. 140)).

ndnSIM Stack Configuration and Installation *

From l. 300 (l. 145), the NDN stack is configured using an object of the ndnSIM `StackHelper` class. As for a CS strategy, *Least Recently Used* (LRU) is chosen on l. 302 (l. 147). The CS is only activated on the intermediate nodes as well as on the zone supernodes: To effectively disable the CS on the UE nodes, the minimum possible CS size of 1 packet is configured on the StackHelper object, before the NDN stack is installed on all UE nodes on l. 307 – 310 (l. 151 – 155). For all other nodes (including the zones' intermediate and supernodes), a default CS size of `csSize` $= 10$ packets is set, before the NDN stack is installed on them (l. 312 – 317 (l. 157 – 162)). The basic configuration of the NDN-layer routing is carried out on l. 320 – 323 (l. 165 – 168): The default chosen

routing strategy is *Best Route* (`best-route`); In the NDN & Chord scenario the routing strategy is actually not relevant, as the NDN communication does not exceed the last-mile zones with simple topology. Thus, *Best Route* is chosen as an arbitrary default strategy for the NDN & Chord scenario. Using an ndnSIM `GlobalRoutingHelper` object, the ndnSIM global routing interface is installed on all nodes, providing all simulated nodes with NDN routing capabilities. For the *Best Route* routing strategy to function properly, NDN routers need information about which NDN prefixes are produced on which nodes, such that interests can be routed to the appropriate lowest-cost next-hop interface: This is realized during the configuration of the ndnSIM Producer applications, as described in the following subsection.

ndnSIM Application Configuration

From l. 326, the configuration of ndnSIM Consumer and Producer applications is carried out. To set up the `ChordProducer` applications, a helper object is instantiated and the default prefix `NDN_PREFIX` = ”/payload” as well as the data item payload size of 1500 B are configured (l. 326 – 328). It follows the setup of the `ConsumerZipfMandelbrot` applications, for which – too – a helper object is instantiated (l. 331), while the following attributes are configured (l. 332 – 337):

- NDN Prefix: `NDN_PREFIX` = ”/payload”
- Frequency: `interestsPerS` = 10
- Randomize: ”uniform”
- Zipf-Mandelbrot parameter of power: `s` = 1.2
- NumberOfContents: `noDataItems` = 100

Thereby, setting ”uniform” on the ”Randomize” attribute causes the effect that consumers will choose inter-interest intervals drawn randomly from a uniform distribution between (0, 1 / Frequency) [27, 28]. The `s` attribute corresponds to the parameter of power to be used in the Zipf-Mandelbrot function. By iterating through all the last-mile zones, the `ChordProducer` and `ConsumerZipfMandelbrot` applications are installed on the UE nodes as well as supernodes (l. 340 – 354): For each supernode, a pointer to its `ChordIpv4` application is retrieved, which gets passed as an attribute to the individual `ChordProducer` applications (l. 343, 344). To enable *Best Route* routing, each supernode is registered on the `GlobalRoutingHelper` object as a producer for prefix `NDN_PREFIX` (l. 345): As soon as all producers have been registered, the `CalculateRoutes` function of ndnSIM's `GlobalRoutingHelper` is invoked, which automatically populates FIBs on all NDN router nodes. During each iteration (*i.e.*, for each last-mile zone), a `ChordProducer` application is installed on the supernode (l. 346), while several `ConsumerZipfMandelbrot` applications are installed on the zone's UEs (l. 351). Scheduling-wise, all ndnSIM applications are started at `NDN_APPS_START_TIME_S`, which corresponds to the point in simulation time after the initialization of Chord (l. 347, 352).

Simulator Instructions and Tracing *

Finally, the scripts define a call to the NS-3 `Simulator::Stop` function, with which the total amount of simulation runtime, *i.e.*, `simulationRuntimeM` = 10 min is fixed (l. 359 (l. 202)): In case of the NDN & Chord scenario, the runtime is prolonged by `NDN_APPS_START_TIME_S` seconds, such that after the Chord layer initialization phase, the ndnSIM apps can run for the same amount of simulated time as in the plain NDN reference scenario.

To collect performance statistics during simulations, two types of tracing systems are set up:

- **Application-level tracing**: Globally installing the ndnSIM `AppDelayTracer` system allows for collecting ndnSIM-application-specific delay statistics and exporting them to a tab-separated file (l. 367 (l. 210)). More specifically, `AppDelayTracer` measures for every interest packet sent the total delay between first sending out the interest and eventually receiving the data.
- **Transit network load tracing**: A custom solution was developed to periodically measure the total network load of the transit network, using the NS-3 tracing subsystem [4]: On l. 205 & 206 (l. 99 & 100), the "PhyTxEnd" trace source of every NetDevice involved in a `PointToPoint` link part of the transit network is connected to the custom trace sink function `TraceTransitLoad` defined on l. 117 – 132 (l. 20 – 35). The "PhyTxEnd" trace fires whenever a packet has been fully transmitted over a channel and delivers the transmitted packet (including its size) as a parameter to the callback function [8]; As the `TraceTransitLoad` function receives callbacks from all transit network NetDevices, it can effectively measure the periodic transit network throughput. This is achieved as follows: When invoked, function `TraceTransitLoad` determines the current simulation period, in tens of seconds (l. 120, 121 (l. 23, 24)); Furthermore, the size of the traced packet is determined and added up in terms of partial network load to a C++ STL `map`, which holds the average network load in Bps per simulation period (l. 122 – 130 (l. 25 – 33)). At the end of the scripts, this map is manually exported to a comma-separated file (l. 373 – 380 (l. 216 – 223)).

4.7.2 Plain NDN Reference Simulation Scenario

As stated earlier, a large amount of code in script `ndn-zones-buckets-small-world.cc` (*cf.* subsection A.3.2) is identical to the the NDN & Chord scenario script: All of the subsections in subsection 4.7.1 denoted with an asterisk (*) contain descriptions concerning both scripts. This section only documents the remaining, fundamentally different parts of the plain NDN reference script.

ndnSIM Producer Application Configuration

From l. 171, the configuration of ndnSIM Producer applications is carried out. Per default, this plain NDN reference scenario deploys standard ndnSIM Producer applications (*cf.* section 4.1.2), which are configured by setting a data item payload size of 1500 B using a helper object (l. 171, 172). Each supernode's Producer is configured to serve a different prefix, such as `/producer3` (l. 187 – 189, *cf.* section 4.6.1). Each Producer as well as its individual prefix are registered with `GlobalRoutingHelper`, such that routing nodes will be able to determine where to forward interests to (l. 190). All individually set up Producer applications are then installed on the supernodes (l. 191).

ndnSIM Consumer Application Configuration: ConsumerZipfBuckets

On l. 175 – 182, ndnSIM Consumer applications are configured using a helper object. The chosen type is `ConsumerZipfBuckets` (*cf.* section 4.6.1). The following attributes are configured (l. 176 – 182):

- NDN Prefix: `NDN_PREFIX` = "/producer"
- Frequency: `interestsPerS` = 10
- NumberOfPrefixBuckets: `NO_ZONES` = 5
- Randomize: "uniform"
- Zipf-Mandelbrot parameter of power: `s` = 1.2
- NumberOfContents: `noDataItems` = 100

The configured no. of prefix buckets corresponds to the simulated five last-mile zones, the remaining attributes are set to the same values as in the `ConsumerZipfMandelbrot` application in the NDN & Chord script (*cf.* section 4.7.1).

By iterating through all last-mile zones, the `ConsumerZipfBuckets` application is installed on all zone UEs (l. 193 – 196).

Forwarding Configuration

Best Route Strategy Per default, the plain NDN script globally installs the *Best Route* forwarding strategy on all nodes. Compared to NDN & Chord, each supernode (*resp.*, producer) originates a different prefix, thus, by iterating through all zones, each producer and its individual prefix are registered on the `GlobalRoutingHelper` object (l. 190): Once all producers have been registered, a call to the `CalculateRoutes` routine causes FIB tables to be populated on all nodes which contain the shortest (*resp.*, 'best') routes towards the producers of individual name prefixes (l. 199).

Flooding Strategy With only a small set of modifications, it is possible to change the routing configuration to cause NDN nodes to broadcast (*resp.*, flood) interest packets towards all upstream neighbours:

- Firstly, the `multicast` strategy must be chosen as the globally installed strategy, instead of `best-route` (l. 165). Thereby, the strategy is set for the entire root prefix namespace ("/"), which comprises all the hierarchically lower prefixes originated by producers, *e.g.* `/producer2`:

```
ns3::ndn::StrategyChoiceHelper::InstallAll ("/",
↪  "/localhost/nfd/strategy/multicast");
```

- Secondly, to prevent from populating FIB holding entries towards individual producers, the `GlobalRouter` interface has to be deactivated. This is achieved easiest by removing lines 168, 190 as well as 199: This effectively leads to sparse FIBs, which do not hold any routes towards specific producer prefixes. Thus, every node between a consumer as well as the producer for the requested prefix will forward interest packets to all its neighbouring upstream nodes, which allows for the simulation of interest flooding on the transit network.

Chapter 5

Evaluation

In this chapter, the results from conducted simulation experiments are presented and discussed. The proposed NDN & Chord architecture (*cf.* Figure 4.1) gets compared against the plain NDN reference architecture (*cf.* Figure 4.7) in terms of the following evaluation criteria:

- *Retrieval Delays*: By comparing the average retrieval delays per data item, the architectures are assessed and compared in terms of users' Quality-of-Experience (QoE), *cf.* section 5.3.
- *Transit Network Load*: In section 5.4, different scenarios are compared *w.r.t.* the measured amount of traffic carried through the small-world transit network.
- *Production Load*: Both architectures are evaluated in terms of the total caused data production load at producer nodes, as presented in section 5.5.

In relation to the production load dimension, it should be mentioned that during all experiments, no secondary effects were simulated whatsoever: More specifically, independently of the simulated interest packet transmission frequency, all data-producing nodes (*i.e.*, the zone supernodes) immediately process incoming interests, while no processing capacity limit was simulated (as it would exist on real-world nodes). With that being said, the production load was merely measured and is presented quantitatively.

5.1 Simulation Parameterization

5.1.1 Constant Simulation Parameters

To allow for fair comparisons of different simulation scenarios, a set of fixed simulation parameters was defined, as presented in the following table:

Table 5.1: Constant Simulation Parameters

Parameter	Value
No. of last-mile zones, supernodes	5
No. of individual data items	100
Interest packet transmission frequency	10 packets per s
Zipf-Mandelbrot function parameters	`s` = 1.2, `q` = 0.7
Max. Content Store size	10 packets
Content Store replacement policy	Least Recently Used
Data packet payload size	1500 B
Simulation runtime	10 min

The parameter of power part of the Zipf-Mandelbrot function (*cf.* subsection 4.4.2) was increased from its default of 0.7 to 1.2 to assign a higher popularity to data items in the low sequence number range. (The same custom parameter of power was set both on the `ConsumerZipfMandelbrot` as well as `ConsumerZipfBuckets` applications.)

In addition to the above parameters, the performance specification of `PointToPoint` channels used in the last-mile zones and in the transit network (*cf.* subsection 4.2.3, subsection 4.5.2) remains unchanged as well.

In all scenarios, the Content Store cache is only activated on the zones' intermediate nodes as well as on supernodes. The chosen cache replacement strategy, *i.e.*, *Least Recently Used*, ensures that the most frequently requested data items remain cached in nodes' CS.

5.1.2 Scaled Simulation Parameters

The following parameters were scaled up to assess both NDN & Chord as well as the plain NDN reference in increasingly demanding scenarios *w.r.t.* load put on the network:

- `n`: The total no. of mobile consumer nodes (UEs)
- `N`: The size of the transit network in no. of nodes

Increasing `n` has a direct influence on the total frequency of interest packets to be processed in the network: For example, if incrementing `n` from 10 to 20, the total rate of interests originated within the network mounts up from 100 to 200 per s, as with every introduced mobile node, during one second 10 additional interests are generated.

Varying `N` – on the other hand – is expected to have an influence on the performance of inter-domain level communication: The greater the number of transit network nodes, the higher the average number of intermediate nodes (*resp.*, hops) which packets between two non-neighbouring supernodes have to traverse, and, the higher the total load carried within the transit network.

All simulation experiments were carried out in both using a small (`N` = 8), as well as a large (`N` = 32) transit network based on connected small-world graphs, as depicted in

the following two figures. The randomly chosen last-mile zone supernodes are coloured red. For both `N`, the chosen rewiring probability used with the Watts-Strogatz graph generation algorithm was $p = 0.7$, while each vertex in the initial lattice was connected to $k = 3$ nearest neighbours (*cf.* subsection 4.5.1).

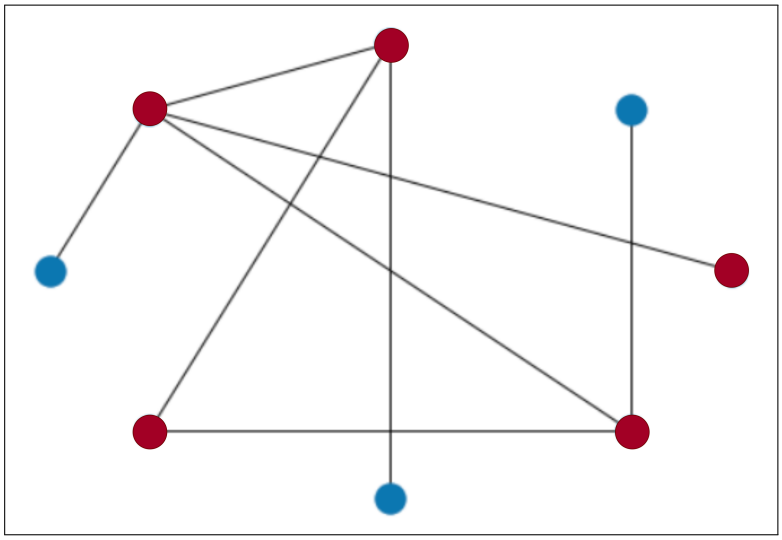

Figure 5.1: Small-World Transit Network (`N` = 8)

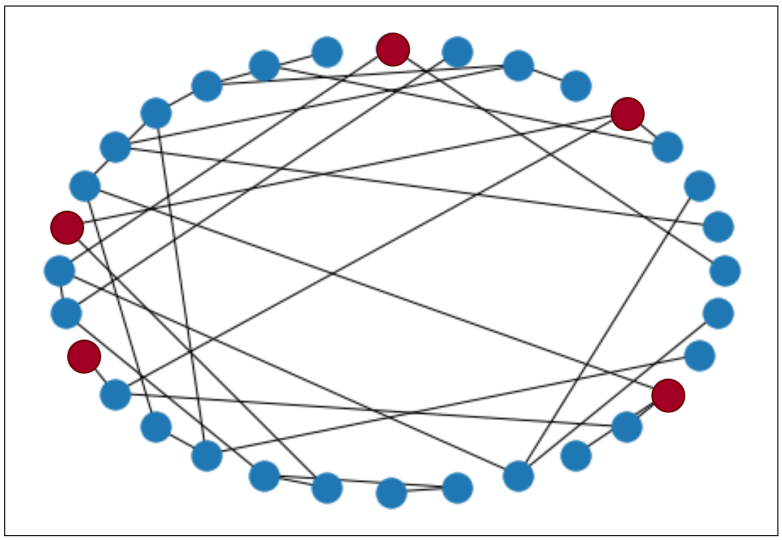

Figure 5.2: Small-World Transit Network (`N` = 32)

5.2 Main Comparison Hypothesis

The proposed NDN & Chord architecture involves two strictly distinct communication layers, which require a means of bridging, to translate from the NDN protocol used at the intra-domain to the Chord protocol at the inter-domain level, and vice-versa. This protocol-translation introduces additional overhead in the form of time delay, compared to native NDN scenarios working under ideal conditions: If a scenario is based on the plain NDN architecture and every NDN node is equipped with a complete and sound FIB routing table, there is a high probability for it to out-perform an NDN & Chord scenario under the same conditions, as not only does the communication remain within the realm of the same technology, but also do complete FIB tables allow for efficient routing using the *Best Route* forwarding strategy.

However, as discussed in subsection 4.6.2, the larger a native NDN network, the less realistic is the assumption, that *Best Route* routing can be globally deployed, thus, *Multicast* routing must be used instead: It therefore seems reasonable to compare plain NDN scenarios with *Multicast* routing against NDN & Chord scenarios: Since *Multicast* forwarding necessarily leads to a higher load in the transit network and less efficient routing, NDN & Chord – due to the low-complexity routing in the Chord overlay – has the potential to perform better than a native NDN network. This effect should also become stronger, the less complete (*resp.*, the sparser) NDN FIBs are in plain NDN scenarios, which leads to strenuous interest flooding on the inter-domain level.

Therefore, the central hypothesis *w.r.t* the comparison of both architectures is that plain NDN with *Flooding* forwarding performs worse than NDN & Chord, since the former causes a significant amount of load on the inter-domain level, leading to lower network efficiency: NDN & Chord – although bridging two distinct technologies – should cause less load within the transit network, leading to more efficient inter-domain level communication as well as faster data retrieval delays.

5.3 Retrieval Delay Comparison

This section covers the results from various simulation experiments, in which the focus was put on the QoE from consumer nodes' perspective, *i.e.*, the full data retrieval delay in-between issuing NDN interest packets and receiving the corresponding data. The presented result plots indicate for each of the 100 simulated data items (*cf.* x-axis) the globally measured minimum, average as well as maximum full retrieval delays: Thereby, the minimum and maximum delays are displayed in terms of a vertical error bar, while the average delays are connected by a curve.

In subsection 5.3.1, NDN & Chord is illustrated alongside plain NDN with *Best Route* routing, for transit network sizes `N` = 8 and `N` = 32, while for the same two transit networks, NDN & Chord is presented in comparison to plain NDN with *Flooding* routing in subsection 5.3.2. For fair comparison, all experiments are based on scenarios with 40 UEs per last-mile zone, which with five domains corresponds to a total no. of `n` = 200 mobile consumer nodes.

For better interpretability, the below table indicates the ranges of the average retrieval delays of the eight presented experiments:

Table 5.2: Average Retrieval Delay Ranges per Experiment

Scenario	**Forwarding Strategy**	`N`	**Avg. Delay Range**	Shortcut
NDN & Chord	*Best Route*	8	37.77 – 106.27 ms	Figure 5.3
Plain NDN			34.12 – 61.38 ms	
NDN & Chord	*Best Route*	32	44.44 – 222.39 ms	Figure 5.4
Plain NDN			37.75 – 113.26 ms	
NDN & Chord	*Best Route*	8	37.77 – 106.27 ms	Figure 5.5
Plain NDN	*Flooding*		88.10 – 408.08 ms	
NDN & Chord	*Best Route*	32	44.44 – 222.39 ms	Figure 5.6
Plain NDN	*Flooding*		132.58 – 471.04 ms	

5.3.1 NDN & Chord vs. plain NDN with *Best-Route* Forwarding

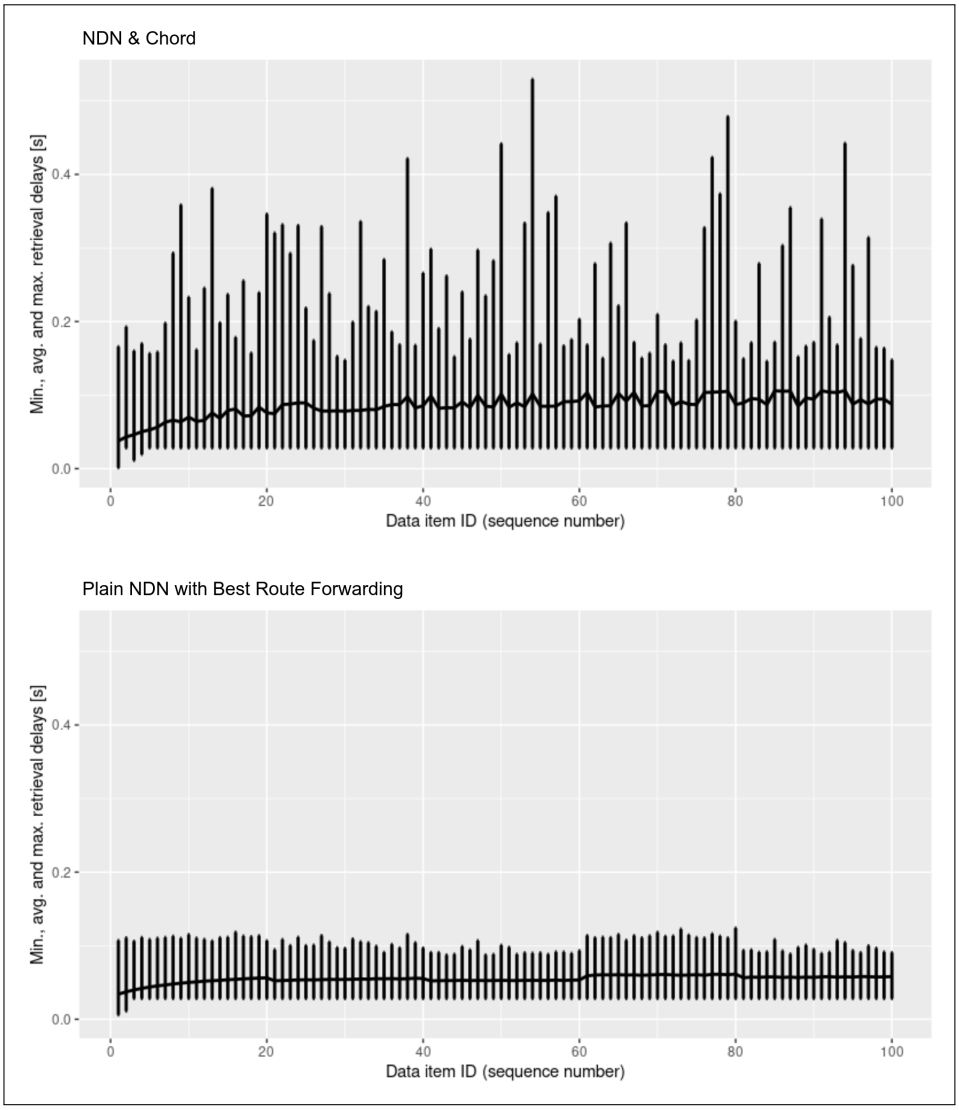

Figure 5.3: NDN & Chord *vs.* Plain NDN with *Best Route* Forwarding ($\mathtt{N} = 8$, $\mathtt{n} = 200$)

Discussion

As expected, in comparison to plain NDN with *Best Route* forwarding, NDN & Chord performs slower due to overhead generated by bridging between NDN and Chord. At the same time, various max. retrieval delay outliers (*e.g.*, 1.07 s for item 78 in Figure 5.4) indicate the inferiority of Chord routing compared to *Best Route* in plain NDN: This can be attributed to the fact that even if Chord guarantees for low-complexity routing on the overlay, one hop from one VNode to another might correspond to underlay traffic which actually has to traverse multiple physical nodes, causing increased delay. Plain NDN with *Best Route* forwarding – on the other hand – has a clear advantage as all globally all traffic remains within the same technology *resp.* networking protocol and, due to the availability of complete FIB tables, each node can forward traffic towards one single lowest-cost next

hop node. In both plain NDN plots one can observe at the varying max. delays that certain data item ranges originate from more or less remote producer nodes (*e.g.*, items 61 – 80 originate from the fourth zone supernode).

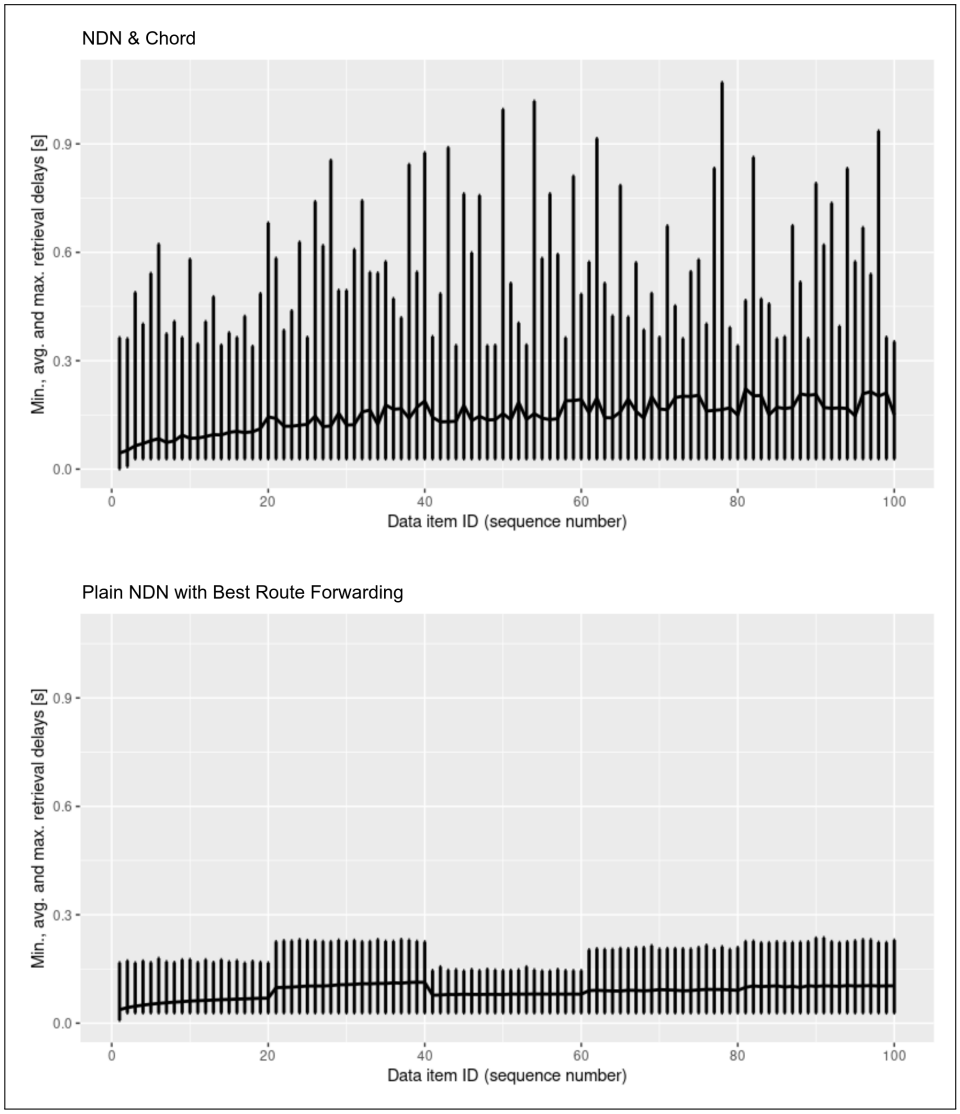

Figure 5.4: NDN & Chord *vs.* Plain NDN with *Best Route* Forwarding (`N` = 32, `n` = 200)

Independently of the transit network size and architecture, one can observe the effect of NDN CS caching: For roughly the first 20 sequence numbers (*i.e.*, the most frequently requested items), the avg. retrieval delay approximately grows linearly, while for higher, less popular data items (which have a low probability of getting cached), avg. delays enter a plateau.

Increasing `N` from 8 to 32 transit nodes leads to roughly doubled avg. retrieval delays for both architectures, however, the observations remain the same.

5.3.2 NDN & Chord vs. plain NDN with *Flooding* Forwarding

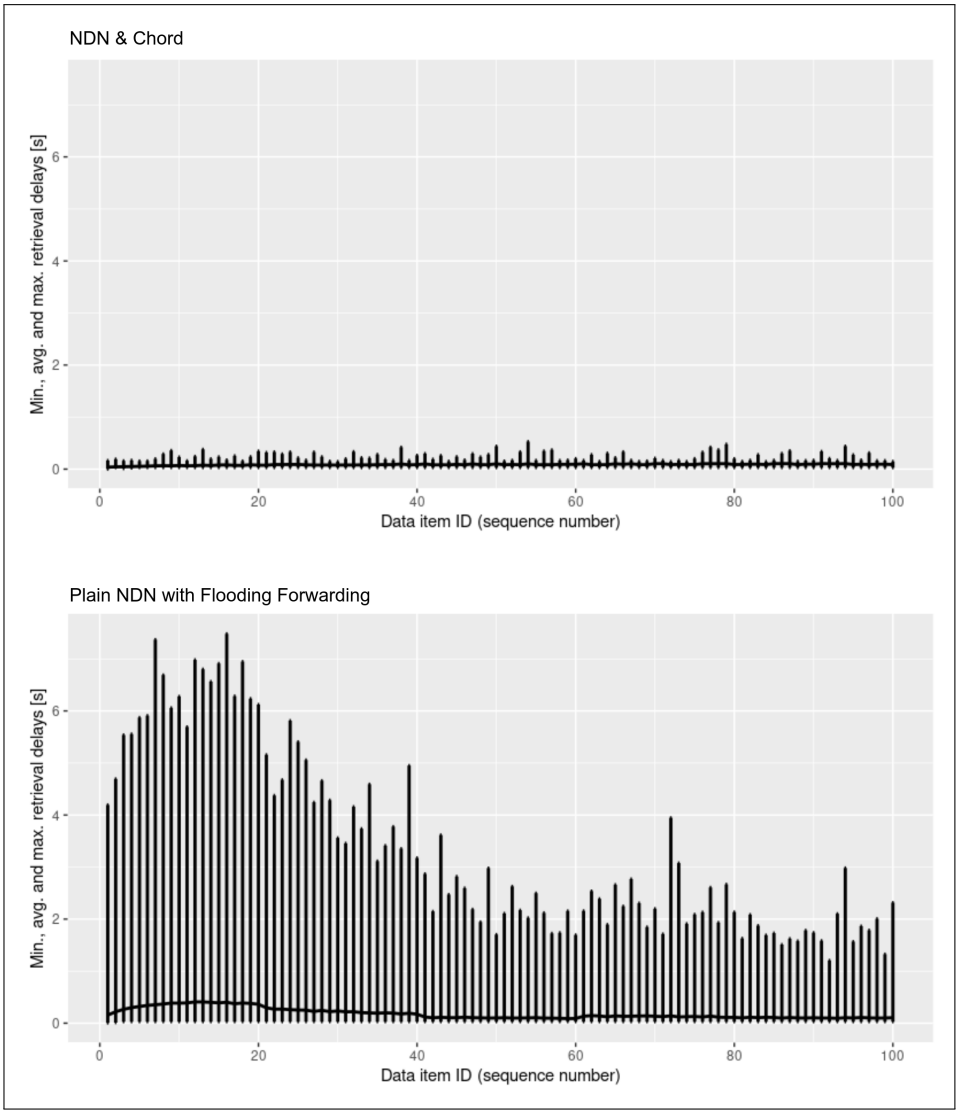

Figure 5.5: NDN & Chord *vs.* Plain NDN with *Flooding* Forwarding ($\mathtt{N} = 8$, $\mathtt{n} = 200$)

Discussion

Introducing interest *Flooding* in plain NDN (*cf.* section 4.7.2) has highly detrimental impacts on the QoE: Compared to the *Best Route* scenarios, with *Flooding* forwarding deployed on all nodes, each interest packet is multiplied according to the number of the nodes' upstream neighbours, which causes a severe amplification of NDN traffic; This effect is intensified, the more nodes lie between consumer and producer nodes, and the more links are attached to intermediate routing nodes. Amplified NDN traffic caused by interest flooding severely strains the transit network (*cf.* section 5.4), such that in some cases data responses take extensive delays to arrive at consumer nodes: In many cases consumer applications time out on non-answered interests and re-send them, which results

in an even higher interest traffic load as well as longer full delays between first issuing an interest and receiving the according data.

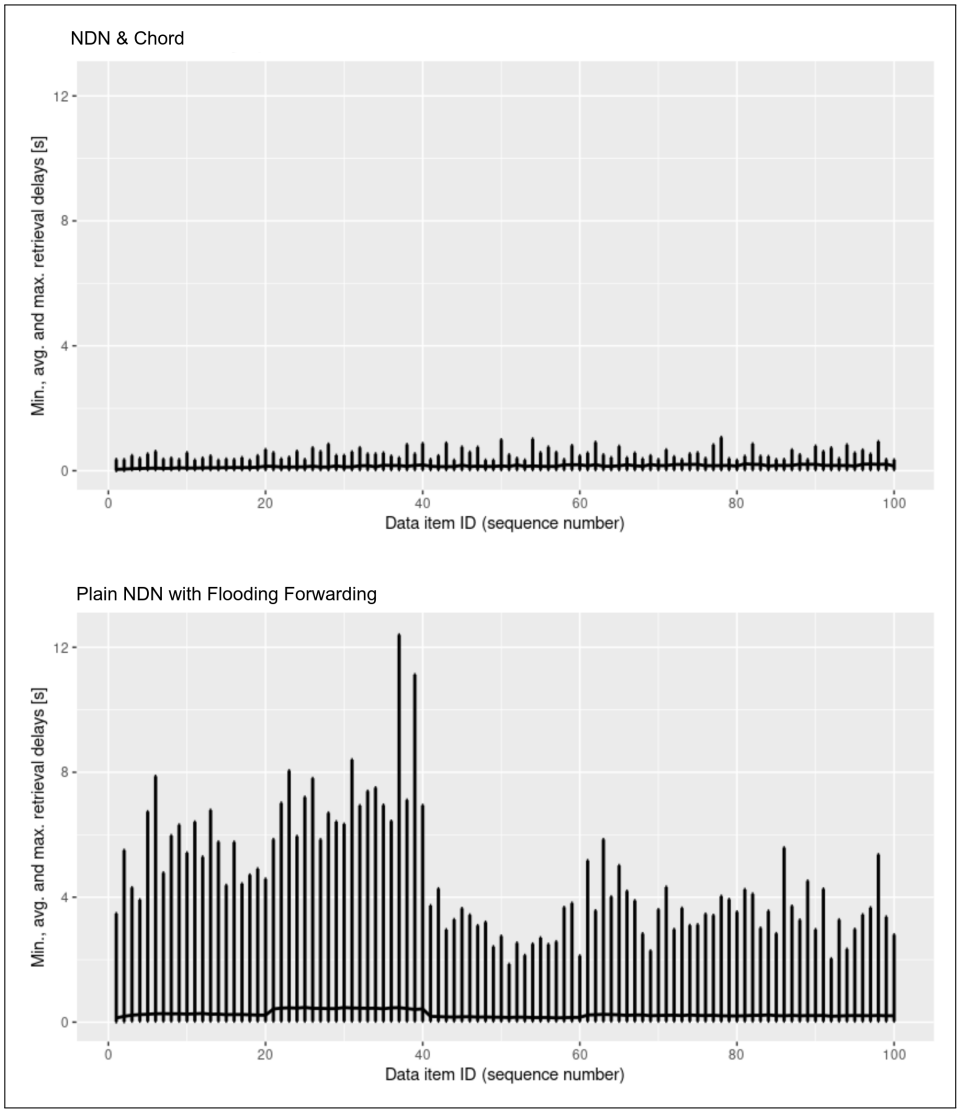

Figure 5.6: NDN & Chord *vs.* Plain NDN with *Flooding* Forwarding (`N` = 32, `n` = 200)

Studying the rising shape of the avg. retrieval curves for low sequence numbers in plain NDN, reveals that CS caching still has a positive impact in terms of lower retrieval avg. delays for popular data items. However, there appear a lot of high max. delays in the low sequence no. ranges: This can be explained by the fact that low sequence no. items get requested at a higher frequency, such that there is a high probability for such retrievals to take place during a state of high network strain, resulting in high retrieval delays. A similar effect occurs for data items originating from nodes which on average are difficult to reach (*i.e.*, a lot of hops between consumer and producer, *e.g.*, data items 21 – 40 for `N` = 32 in Figure 5.6), as more intermediate routing nodes lead to a higher degree of traffic amplification and thus, more strain on the transit network.

For both `N` = 8 as well as `N` = 32, NDN & Chord attains a better QoE in terms of lower average and maximum retrieval delays compared to plain NDN with *Flooding*.

5.4 Transit Network Load Comparison

The bar plots presented in this section indicate the average traffic load caused on the small-world transit network during differently scaled simulation scenarios. The network load was derived by periodically measuring the total throughput on all simulated links which compose the small-world network (*cf.* section 4.7.1): The values indicated by the bars indicate the average network load throughout all simulation periods.

In Figure 5.7, the results from scenarios using the transit network of size $\mathtt{N} = 8$ are presented, while Figure 5.8 displays $\mathtt{N} = 32$.

For both transit network sizes, the number of mobile consumer nodes (*resp.* UEs) is scaled up according to $\mathtt{n} = 5, 10, 25, 50, 100, 150, 200$. Thereby, NDN & Chord is evaluated against plain NDN configured with the *Best Route* as well as *Flooding* forwarding strategies.

To allow for a fair comparison, for NDN & Chord the avg. transit network load only covers the 10 simulated minutes after the initialization of Chord, *i.e.*, the traffic originating from the insertion of VNodes and Chord data objects is not taken into the account during the computation of the total avg. transit network load.

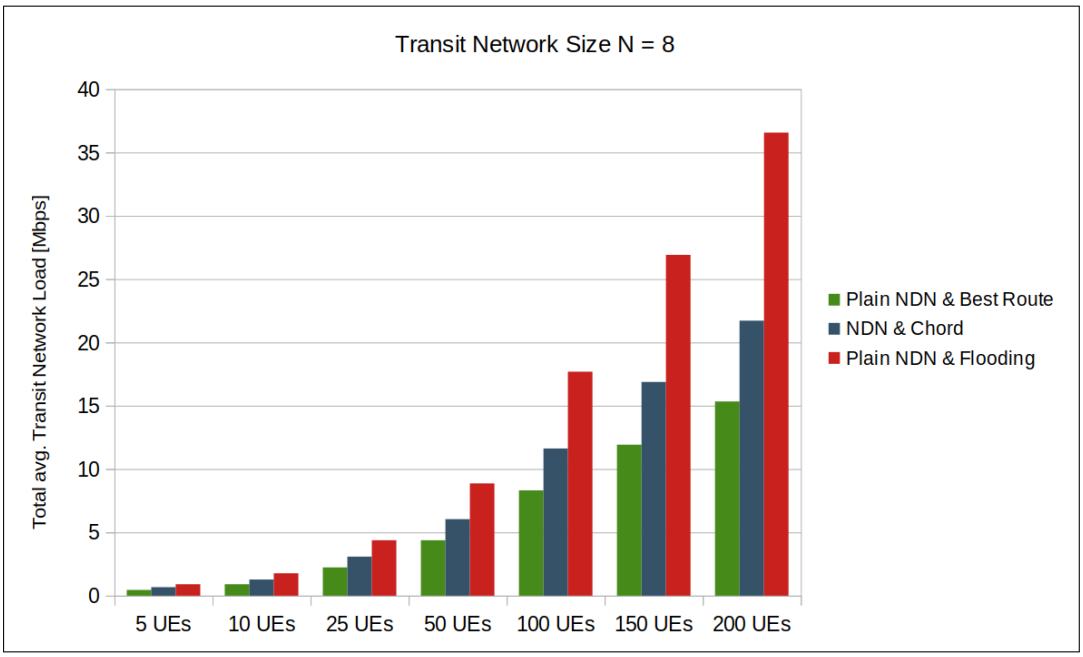

Figure 5.7: Transit Network Load Comparison for $\mathtt{N} = 8$

Discussion

Generally, it can be stated that plain NDN with *Best Route* forwarding is the most efficient strategy, not only delay-wise (*cf.* section 5.3) but also in terms of the lowest generated transit network load for all last-mile zone sizes $\mathtt{n}$ as well as transit network sizes $\mathtt{N}$. On the

other hand, plain NDN with *Flooding* forwarding – as expected – causes a significantly higher transit network load in all scenarios.

When comparing NDN & Chord with plain NDN & *Flooding* for $\mathtt{N} = 8$, NDN & Chord clearly wins in terms of a lower achieved transit network load in all seven scenarios simulating different $\mathtt{n}$.

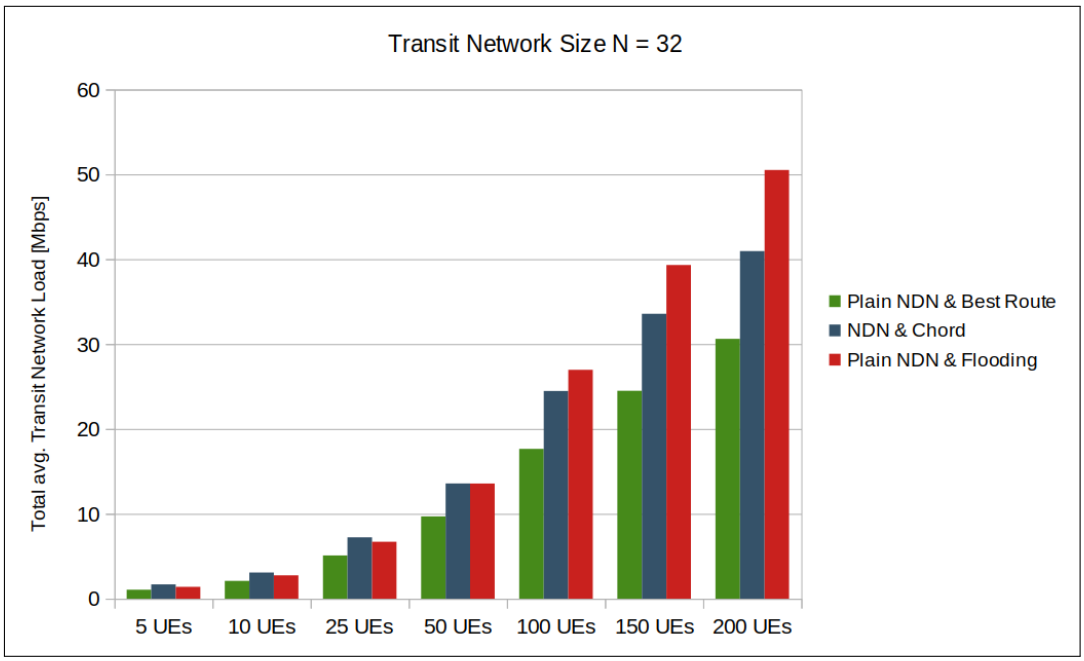

Figure 5.8: Transit Network Load Comparison for $\mathtt{N} = 32$

As discussed in section 5.3.1, increasing the size of the transit network leads to more expensive routing on the Chord overlay. This does not only find expression in increased retrieval delays but also in elevated transit network load with $\mathtt{N} = 32$: For $\mathtt{n} = 5$, 10, 25 up until 50, NDN & Chord even causes a slightly higher transit network load compared to plain NDN with *Flooding*:

However, the results of scenarios for larger $\mathtt{n}$ (*i.e.*, >50) support the main hypothesis, as NDN & Chord clearly takes the lead in putting less strain on the transit network compared to plain NDN & *Flooding*, which with increasing $\mathtt{n}$ causes more and more NDN control traffic amplification.

5.5 Production Load Comparison

To cover the third and last evaluation dimension, the focus is laid on the total production load occurring at all data producing nodes during the entire simulations. More specifically, production load refers to the total number of times a zone supernode has processed an incoming interest packet for an uncached data item and issued a data packet towards downstream nodes. In the NDN & Chord architecture, one data item production additionally comprises a retrieval on the DHT overlay (*cf.* Figure 4.5), while for plain NDN scenarios, producer nodes merely create and send virtual payload packets: Clearly, different data production approaches can lead to different computational effort occurring at producer nodes. However, this work only considers quantitative production load, a comparison of the architectures *w.r.t.* computational cost on producer nodes is out-of-scope. The total production load was derived by analyzing NS-3 simulation logs and computing the total no. of times `Producer` applications installed on zone supernodes had issued data packets to downstream nodes using standard Unix programs `grep` and `wc`.

Depending on the simulated architecture, forwarding strategy as well as transit network size, a different Total No. of Productions results, as indicated in the below table. All six results originate from experiments using 40 UEs per last-mile zone, *i.e.*, `n` = 200. With the default parameters for consumer nodes' interest sending frequency as well as the total simulation run time of 10 min, the global total amount of issued interest packets corresponds to 200 UEs * 10 interests per s * 600 s = 1'200'000 interests.

Table 5.3: Total Number of Productions per Experiment (`n` = 200)

Scenario	**Forwarding Strategy**	`N`	**Total No. of Productions**
NDN & Chord	*Best Route*	8	540'251
Plain NDN			391'989
Plain NDN	*Flooding*		489'674
NDN & Chord	*Best Route*	32	458'978
Plain NDN			367'557
Plain NDN	*Flooding*		378'514

Discussion

By comparing the resulting production load for the six different scenarios, one can make the following three observations:

Firstly, for both transit network sizes `N` = 8 as well as `N` = 32, zone supernodes in the NDN & Chord architecture have to produce the highest amount of data items. The reason therefore can be found in the fact that in NDN & Chord, every interest packet for uncached data items is necessarily picked up and processed by a zone supernode, while all NDN traffic remains within last-mile zones and no interest packets pass over into the inter-domain level network whatsoever. This is completely different for the native NDN networks simulated in plain NDN, where supernodes forward interests to the transit network, if they are not responsible for the queried name prefix. Thereby, there is a higher

chance for interests to be served from a CS cache on a foreign zone's supernode, compared to NDN & Chord, in which interests only ever encounter two nodes with activated CS, *i.e.*, the local intermediate as well as supernode (*cf.* subsection 5.1.1): The smaller the probability for interests to be served from CS caches, the higher is the quantitative load accumulating at producer nodes, which explains the high figures for NDN & Chord. However, in NDN & Chord, producer nodes still only have to process approximately up to 45% of the total 1.2 M interests, while the rest can be directly served from local CS caches.

Secondly, for both $\mathtt{N} = 8$ as well as $\mathtt{N} = 32$, a higher production load results in plain NDN scenarios using *Flooding*, compared to the *Best Route* forwarding strategy. This can be attributed to interest amplification caused by forwarding nodes broadcasting interest packets towards all upstream nodes, which leads to a generally higher frequency of incoming interests at producer nodes across the entire transit network.

Thirdly, for all three scenarios with $\mathtt{N} = 32$, the total no. of produced data items is smaller compared to $\mathtt{N} = 8$. In case of NDN & Chord, this is due to the more complex structure of the transit network, leading to more expensive overlay routing and ultimately, slower data retrievals and productions. For similar reasons, there is less production load in both plain NDN scenarios, since on average, it takes a longer delay for interest packets to reach producer nodes when having to surpass the larger transit network and, the probability is higher for interests to be served from some intermediate CS cache before reaching a producer, in contrast to $\mathtt{N} = 8$.

Chapter 6

Summary and Conclusions

The contributions of this work are threefold:

Firstly, a layered network architecture for content distribution was designed, which combines ICN and DHT technologies: In terms of ICN, the architecture foresees an intra-domain layer consisting of several mobile last-mile networks operating on NDN – a future internet architecture. Thereby, the last-mile zones were defined according to the typical hierarchical structure of mobile networks, while the assumption was made that all mobile devices are consumers of data, which express their aim to retrieve named data towards hierarchically superior network participants, *i.e.*, to supernodes part of every last-mile domain. In terms of an inter-domain communication layer, all supernodes were defined to be involved in a Chord DHT, which is used for the distributed storage as well as delivery of data items among different zones *resp.*, domains: Compared to the NDN-based intra-domain layer, the Chord protocol used on the inter-domain level operates as an overlay on top of the traditional IP-based internet. This provides the advantage that in spite of the lack of a global native NDN network, the benefits of NDN can be leveraged within several independent domains interconnected over existing internet infrastructure.

Secondly, the proposed NDN & Chord architecture was implemented in terms of a simulation in the NS-3 network simulator framework, by incorporating the state-of-the-art simulator modules ndnSIM as well as ns-3-chord, to model the ICN and DHT communication layers. To facilitate realistic network simulations at the small scale, the simulation was configured according to three real-world aspects *resp.*, models: For simulating the performance characteristics of mobile networks, the communication channels at the intra-domain level were configured according to delay and throughput measurements derived from an NS-3 LTE simulation experiment arranged to assess the efficiency of over-the-air data exchange among mobile devices and base stations. To represent the pattern of the demand for popular information observed in real-world, mobile consumer nodes are modeled to request data according to the Zipf-Mandelbrot distribution, with which consumers query a certain range of data disproportionately often, while other items are only demanded seldomly. Lastly, to model the transit network at the inter-domain level (*i.e.*, among router nodes and last-mile zone supernodes), random connected small-world networks were generated using the Watts-Strogatz algorithm, which embody the same characteristics as real-world internet backbone networks used at the inter-operator level.

Ultimately, to allow for an assessment and fair comparison of the suggested NDN & Chord architecture, a native NDN reference architecture (denoted as *plain NDN*) was designed, in which NDN is deployed as the sole global communication technology, *i.e.*, also on the inter-domain level.

Thirdly, the suggested NDN & Chord architecture was evaluated and compared with the plain NDN reference by conducting different simulation experiments: Analyzing the QoE achieved from a consumer node perspective, *i.e.*, the average data item retrieval delays, has revealed that in general NDN & Chord performs slower than plain NDN under perfect routing conditions. However, in scenarios in which plain NDN operates on sparse FIB tables and *Multicast* instead of *Best Route* forwarding, it could be shown that NDN & Chord clearly operates faster and more reliably than plain NDN under aggravated routing conditions. Not only *w.r.t.* QoE, but also in terms of the traffic load put upon the transit network, NDN & Chord scores second, in-between plain NDN with *Best Route* forwarding causing the least load, as well as plain NDN with *Flooding* forwarding straining the transit network the most. Lastly, NDN & Chord was compared to plain NDN scenarios while focusing on the total quantitative production load generated at producer nodes: The finding of this analysis is that in NDN & Chord, producer nodes have to manage a higher load compared to native NDN scenarios, since as opposed to plain NDN, NDN traffic does not exceed local domains (*resp.*, last-mile zones), such that all interest packets have to be processed by local supernodes.

In summary, one can conclude that the introduced NDN & Chord architecture has proven itself as a well performing option when it comes to interconnecting several independent NDN domains, especially if per the lack of infrastructure, a large native multi-domain NDN network cannot be realized, or, if NDN is operating on poor routing conditions.

6.1 Future Work

It has been discussed in section 5.5 that compared to plain NDN with the *Best Route* as well as *Flooding* forwarding strategies, in NDN & Chord scenarios producer nodes have to serve data items at a higher frequency. As pointed out, in the experiments carried out in the scope of this work, simulated producer nodes do not have a capacity limit and thus, accept any frequency of incoming interest packets. As part of a future contribution, the analysis of production load could be expanded as follows:

Initially, the required computational effort for producer nodes to bridge NDN and Chord and to retrieve a data item from the DHT shall be studied: This could be accomplished by setting up a testbed in which on a real-world host, the combined resource consumption is measured for receiving and translating an NDN interest into a Chord retrieval, Chord overlay routing to determine the data object owner, downloading the queried data object as well as encapsulating the data into an NDN packet to be sent back to the downstream. Once a resource consumption reference has been identified, it will be possible to place it into relation with the maximum manageable load of one single host, to define a production bottleneck in terms of a maximum frequency of incoming interests: This will allow for conducting more accurate experiments which involve the simulation of limited producer node capacity, in which interest packets are discarded, if they are directed to an overloaded supernode. If in plain NDN scenarios, too, realistic max. producer capacity is simulated, it would be possible to re-assess native NDN against the layered NDN & Chord approach, to study the implications of different quantitative production loads towards the efficiency of different networks.

Bibliography

[1] Google Lens Image Recognition Application. https://lens.google. Accessed: 2021-07-20.

[2] NDN project team: "Named Data Networking (NDN) - A Future Internet Architecture. http://named-data.net. Accessed: 2021-06-28.

[3] NDN Testbed. https://named-data.net/ndn-testbed. Accessed: 2021-07-06.

[4] NS-3 Network Simulator Manual (Version 3.29). https://www.nsnam.org/docs/release/3.29/manual/singlehtml. Accessed: 2021-07-24.

[5] NS-3 Network Simulator Model Library (Version 3.29). https://www.nsnam.org/docs/release/3.29/models/singlehtml. Accessed: 2021-07-24.

[6] NS-3 Network Simulator Tutorial (Version 3.29). https://www.nsnam.org/docs/release/3.29/tutorial/singlehtml. Accessed: 2021-07-24.

[7] NS-3 UdpClientServer Applications. https://www.nsnam.org/doxygen/group__udpclientserver.html. Accessed: 2021-07-26.

[8] NS-3 Version 3.29. https://www.nsnam.org/releases/ns-3-29. Accessed: 2021-07-24.

[9] "Get NFD Connected": NDN as an Overlay on Top of Traditional IP. https://named-data.net/2015/01/06/get-nfd-connected, Jan. 2015. Accessed: 2021-07-06.

[10] Android OS Vulnerabilities (Cambridge University). http://androidvulnerabilities.org, 2019. Accessed: 2021-06-26.

[11] Android Security Team (Google). https://source.android.com/security/overview/updates-resources, Dec. 2020. Accessed: 2021-06-26.

[12] Lada Adamic and Bernardo Huberman. Zipf's Law and the Internet. *Glottometrics*, 3, 11 2001.

[13] Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnSIM: NDN Simulator for NS-3. Technical Report NDN-0005, NDN, October 2012.

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

[15] Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[16] Torsten Braun, Volker Hilt, Markus Hofmann, Ivica Rimac, Moritz Steiner, and Matteo Varvello. Service-Centric Networking. In *2011 IEEE International Conference on Communications Workshops (ICC)*, pages 1–6, 2011.

[17] Stephen Fagan and Ramazan Gençay. An Introduction to Textual Econometrics. *Handbook of empirical economics and finance*, pages 133–154, 12 2010.

[18] Pascal Felber, Peter Kropf, Eryk Schiller, and Sabina Serbu. Survey on Load Balancing in Peer-to-Peer Distributed Hash Tables. *IEEE Communications Surveys Tutorials*, 16(1):473–492, 2014.

[19] Mikael Gasparyan. *Service-Centric Networking*. PhD thesis, University of Bern, May 2020.

[20] Mikael Gasparyan, Torsten Braun, and Eryk Schiller. L-SCN: Layered SCN Architecture with Supernodes and Bloom Filters. In *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 899–904, 2017.

[21] Harjot Gill. ns-3-chord: Chord/DHash DHT in NS-3. `https://code.nsnam.org/gillh/ns-3-chord`. Accessed: 2021-07-29.

[22] Tim Greene. What is the internet backbone and how it works. `https://www.networkworld.com/article/3532318/what-is-the-internet-backbone-and-how-it-works.html`. Accessed: 2021-08-02.

[23] David Griffin, Miguel Rio, Pieter Simoens, Piet Smet, Frederik Vandeputte, Luc Vermoesen, Dariusz Bursztynowski, Folker Schamel, and Michael Franke. Service-Centric Networking. In *Handbook of Research on Redesigning the Future of Internet Architectures*, pages 68–95. IGI Global, 2015.

[24] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[25] Xiaoke Jiang. ConsumerZipfMandelbrot ndnSIM Application. `https://ndnsim.net/2.7/applications.html#consumerzipfmandelbrot`. Accessed: 2021-07-29.

[26] Benoît Mandelbrot. Information Theory and Psycholinguistics. *B.B. Wolman and E. Nagel (ed.). Scientific psychology*, 1965.

[27] Spyridon Mastorakis, Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnSIM 2: An Updated NDN Simulator for NS-3. Technical Report NDN-0028, Revision 2, NDN, November 2016.

[28] Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. On the Evolution of ndnSIM: an Open-Source Simulator for NDN Experimentation. *ACM Computer Communication Review*, July 2017.

[29] Sergei Nikolaev, Eddy Banks, Peter D. Barnes, David R. Jefferson, and Steven Smith. Pushing the Envelope in Distributed NS-3 Simulations: One Billion Nodes. In *Proceedings of the 2015 Workshop on Ns-3*, WNS3 '15, page 67–74, New York, NY, USA, 2015. Association for Computing Machinery.

[30] David Oran and Dirk Kutscher. Information Centric Networking Research Group. `https://datatracker.ietf.org/rg/icnrg/about`. Accessed: 2021-06-28.

[31] Lawrence G. Roberts. The Evolution of Packet Switching. `https://web.archive.org/web/20160324033133/http://www.packet.cc/files/ev-packet-sw.html`, Nov. 1978. Accessed: 2021-06-28.

[32] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[33] Seth Terashima (Tetra7 (talk)). Chord P2P network (picture). License: CC BY-SA 3.0. `https://commons.wikimedia.org/wiki/File:Chord_network.png`. Accessed: 2021-07-13.

[34] Seth Terashima (Tetra7 (talk)). Chord Route (picture). License: CC BY-SA 3.0. `https://commons.wikimedia.org/wiki/File:Chord_route.png`. Accessed: 2021-07-13.

[35] NDN Project Team. ndn-cxx library (NDN C++ library with eXperimental eXtensions). `https://named-data.net/doc/ndn-cxx/current`. Accessed: 2021-07-26.

[36] Qawi K Telesford, Karen E Joyce, Satoru Hayasaka, Jonathan H Burdette, and Paul J Laurienti. The Ubiquity of Small-World Networks. *Brain connectivity*, 1(5):367–375, 2011.

[37] Duncan J Watts and Steven H Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393(6684):440–442, 1998.

[38] Chengkai Yan, Quang Ngoc Nguyen, Ilias Benkacem, Daisuke Okabe, Akihiro Nakao, Toshitaka Tsuda, Cutifa Safitri, Tarik Taleb, and Takuro Sato. Design and Implementation of Integrated ICN and CDN as a Video Streaming Service. In Marco Di Felice, Enrico Natalizio, Raffaele Bruno, and Andreas Kassler, editors, *Wired/Wireless Internet Communications*, pages 194–206, Cham, 2019. Springer International Publishing.

[39] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named Data Networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, July 2014.

[40] George Kingsley Zipf. Selected Studies of the Principle of Relative Frequency in Language. 1932.

Abbreviations

CDN	Content Delivery Network
CS	Content Store
CSV	Comma-Separated Values (file format)
DHT	Distributed Hash Table
FIB	Forwarding Information Base
ICN	Information Centric Networking
ID	Identifier
IP	Internet Protocol
L-SCN	Layered Service Centric Networking
MNO	Mobile Network Operator
NDN	Named Data Networking
NFD	Named Data Networking Forwarding Daemon
NS-3	Network Simulator version 3
OS	Operating System
OSI	Open Systems Interconnection model
P2P	Peer-to-peer network
PIT	Pending Interest Table
PTP	Point-to-point connection
QoE	Quality-of-Experience
TCP	Transport Control Protocol
TSV	Tab-Separated Values (file format)
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VPN	Virtual Private Network
WWW	World Wide Web

Glossary

Caching to cache refers to storing replicated data in a network node's memory

Downstream refers to the direction of data traffic towards content *resp.*, data consuming nodes in a network

Hashing refers to the process of using a hash function to encode input data of arbitrary size to produce a fixed-size hash (value)

Host/Node Terms *host* and *node* are used interchangeably in this work to denote communication capable devices involved in one or more networks

Overlay describes a type of network which operates as an additional layer on top of another established network technology (*i.e.*, underlay)

Supernode refers to a type of node having the responsibility to bridge the communication between two or more distinct domains part of a larger network

Upstream refers to the direction data traffic towards content *resp.*, data producing nodes in a network

List of Figures

List of Tables

Appendix A

Source Code

A.1 LTE-EPC Performance Measurements

The following source code originates from C++ NS-3 simulation script called `lena-performance-measurements.cc`, which was used to determine realistic LTE performance measurements, as described in subsection 4.2.2.

```
1  #include "ns3/lte-helper.h"
2  #include "ns3/epc-helper.h"
3  #include "ns3/core-module.h"
4  #include "ns3/network-module.h"
5  #include "ns3/ipv4-global-routing-helper.h"
6  #include "ns3/internet-module.h"
7  #include "ns3/internet-apps-module.h"
8  #include "ns3/mobility-module.h"
9  #include "ns3/lte-module.h"
10 #include "ns3/applications-module.h"
11 #include "ns3/flow-monitor-module.h"
12 #include <vector>
13 #include <cstdio>
14 #include <iostream>
15
16
17 using namespace ns3;
18
19 /**
20  * This script can be used to measure the performance of LTE+EPC.
21  * In each independent LTE network, one eNodeB is installed.
22  * To collect LTE performance measurements, we work with one LTE network and
    ↪  install UDP client
23  * applications on the UEs, as well as a UDP server on the corresponding
    ↪  eNodeB.
24  *
```

```
25  * At a steady periodicity, the client applications on the UEs issue UDP
    ↪  packets of a fixed payload size
26  * towards the server application on the eNodeB.
27
28  * Finally, using the FlowMonitor module we collect several performance
    ↪  statistics on the UDP package exchange.
29  */
30
31 NS_LOG_COMPONENT_DEFINE ("EpcPerformanceMeasurement");
32
33
34 int
35 main (int argc, char *argv[])
36 {
37   //
38   // Enable logging for UdpClient and UdpServer
39   //
40   LogComponentEnable ("UdpClient", LOG_LEVEL_INFO);
41   LogComponentEnable ("UdpServer", LOG_LEVEL_INFO);
42
43   uint16_t numberOfNetworks = 1;
44   uint16_t numberOfNodes = 5;
45   double distance = 500.0;
46
47   InternetStackHelper internet;
48   Ipv4StaticRoutingHelper ipv4RoutingHelper;
49
50   std::vector<Ptr<LteHelper>> IndependentNetworks (numberOfNetworks);
51   std::vector<Ptr<PointToPointEpcHelper>> IndependentEpcs (numberOfNetworks);
52   std::vector<Ptr<Node>> IndependentPgws (numberOfNetworks);
53   std::vector<Ipv4InterfaceContainer> IndependentInterface (numberOfNetworks);
54   std::vector<NodeContainer> IndependentueNodes (numberOfNetworks);
55   std::vector<NodeContainer> IndependentenbNodes (numberOfNetworks);
56   std::vector<Ptr<ListPositionAllocator>> IndependentPositionAllocators
     ↪  (numberOfNetworks);
57   std::vector<MobilityHelper> IndependentMobilityHelpers (numberOfNetworks);
58   std::vector<NetDeviceContainer> IndependentueDevices (numberOfNetworks);
59   std::vector<NetDeviceContainer> IndependentenbDevices (numberOfNetworks);
60   std::vector<Ipv4InterfaceContainer> IndependentueIpIfaces (numberOfNetworks);
61
62   for (uint16_t nn = 0; nn < numberOfNetworks; nn++)
63     {
64
65       Ptr<LteHelper> &lteHelper = IndependentNetworks[nn];
66       lteHelper = CreateObject<LteHelper> ();
67
68       Ptr<PointToPointEpcHelper> &epcHelper = IndependentEpcs[nn];
69
70       // s1-u interface on 2.(nn).0.0, mask defined in
         ↪  src/lte/helper/point-to-point-epc-helper.cc
```

```
71        std::ostringstream s1ubaseaddress;
72        s1ubaseaddress << "2." << nn << ".0.0";
73        // x2 interface on 3.(nn).0.0, mask defined in
          ↪ src/lte/helper/point-to-point-epc-helper.cc
74        std::ostringstream x2baseaddress;
75        x2baseaddress << "3." << nn << ".0.0";
76        // ue base interface 4.(nn).0.0, mask defined in
          ↪ src/lte/helper/point-to-point-epc-helper.cc
77        std::ostringstream uebaseaddress;
78        uebaseaddress << "4." << nn << ".0.0";
79        epcHelper = CreateObject<PointToPointEpcHelper> (s1ubaseaddress.str
          ↪ ().c_str (),
80                                                         x2baseaddress.str
                                                           ↪ ().c_str (),
81                                                         uebaseaddress.str
                                                           ↪ ().c_str ());
82
83        lteHelper->SetEpcHelper (epcHelper);
84
85        Ptr<Node> &pgw = IndependentPgws[nn];
86        pgw = epcHelper->GetPgwNode ();
87
88
89        NodeContainer &ueNodes = IndependentueNodes[nn];
90        NodeContainer &enbNodes = IndependentenbNodes[nn];
91
92        enbNodes.Create (1); // Create one eNB per zone
93        ueNodes.Create (numberOfNodes);
94
95        // Install Mobility Models
96        Ptr<ListPositionAllocator> &enbPositionAlloc =
          ↪ IndependentPositionAllocators[nn];
97        enbPositionAlloc = CreateObject<ListPositionAllocator> ();
98        Ptr<ListPositionAllocator> uePositionAlloc =
          ↪ CreateObject<ListPositionAllocator> ();
99
100       enbPositionAlloc->Add (Vector (0, 0, 0)); // Place single zone eNB to
          ↪ point of origin
101
102       for (uint16_t i = 1; i <= numberOfNodes; i++)
103       {
104         uePositionAlloc->Add (Vector (distance * i, 0, 0));
105       }
106
107       MobilityHelper &mobility = IndependentMobilityHelpers[nn];
108       MobilityHelper ueMobility;
109
110       mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
111       ueMobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
112       mobility.SetPositionAllocator (enbPositionAlloc);
```

```
113       ueMobility.SetPositionAllocator (uePositionAlloc);
114       mobility.Install (enbNodes);
115       ueMobility.Install (ueNodes);
116
117       // Install LTE Devices to the nodes
118       NetDeviceContainer &enbLteDevs = IndependentenbDevices[nn];
119       enbLteDevs = lteHelper->InstallEnbDevice (enbNodes);
120
121       NetDeviceContainer &ueLteDevs = IndependentueDevices[nn];
122       ueLteDevs = lteHelper->InstallUeDevice (ueNodes);
123
124       // Install the IP stack on the UEs
125       internet.Install (ueNodes);
126
127       // Assign IP address to UEs
128       Ipv4InterfaceContainer &ueIpIface = IndependentueIpIfaces[nn];
129       ueIpIface = epcHelper->AssignUeIpv4Address (ueLteDevs);
130
131       // default route on UEs
132       for (uint32_t u = 0; u < ueNodes.GetN (); ++u)
133         {
134           Ptr<Node> ueNode = ueNodes.Get (u);
135           // Set the default gateway for the UE
136           Ptr<Ipv4StaticRouting> ueStaticRouting =
137               ipv4RoutingHelper.GetStaticRouting (ueNode->GetObject<Ipv4> ());
138           ueStaticRouting->SetDefaultRoute
      ↪  (epcHelper->GetUeDefaultGatewayAddress (), 1);
139         }
140
141       // Attach all UE to first eNodeB
142       for (uint16_t i = 0; i < numberOfNodes; i++)
143         {
144           lteHelper->Attach (ueLteDevs.Get (i), enbLteDevs.Get (0));
145         }
146     }
147
148   // Point to the eNodeB of the first network
149   Ptr<Node> server = IndependentenbNodes[0].Get (0);
150
151   // Install one UdpServer application on the server node.
152   uint16_t port = 4000;
153   UdpServerHelper udpServer (port);
154   ApplicationContainer udpApps = udpServer.Install (server);
155   udpApps.Start (Seconds (1.0));
156   udpApps.Stop (Seconds (10.0));
157
158   // Install one UdpClient application to send UDP datagrams from the client to
      ↪  the server.
159   uint32_t MaxPacketSize = 1500; // LTE MTU
160   Time interPacketInterval = Seconds (0.05);
```

```
161   uint32_t maxPacketCount = 320;
162   Address serverAddress = Address (server->GetObject<Ipv4> ()->GetAddress (1,
      ↪  0).GetLocal ());
163   UdpClientHelper udpClient (serverAddress, port);
164   udpClient.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
165   udpClient.SetAttribute ("Interval", TimeValue (interPacketInterval));
166   udpClient.SetAttribute ("PacketSize", UintegerValue (MaxPacketSize));
167   for (uint16_t i = 0; i < numberOfNodes; i++)
168   {
169     // Install UdpClient applications on all the nodes of the first network
170     udpApps = udpClient.Install (IndependentueNodes[0].Get (i));
171     udpApps.Start (Seconds (2.0));
172     udpApps.Stop (Seconds (10.0));
173   }
174
175   // Set up Flow monitor
176   Ptr<FlowMonitor> flowMonitor;
177   FlowMonitorHelper flowHelper;
178   flowMonitor = flowHelper.InstallAll();
179
180   Simulator::Stop (Seconds (12));
181   Simulator::Run ();
182
183   // Export flow stats to file
184   flowMonitor->SerializeToXmlFile ("lena-udp-flow-stats.flowmon", false,
      ↪  false);
185
186   Simulator::Destroy ();
187   return 0;
188 }
```

A.2 ChordProducer ndnSIM Application

The source code attached to this section originates from `ChordProducer`, an ndnSIM application which was developed to connect the NDN-based intra-domain layer, with the Chord-based inter-domain communication layer (*cf.* section 4.3).

A.2.1 ndn-chord-producer.hpp

```
1  /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2
3  #ifndef NDN_CHORD_PRODUCER_H
4  #define NDN_CHORD_PRODUCER_H
5
6  #include "ns3/ndnSIM/model/ndn-common.hpp"
7
8  #include "ndn-app.hpp"
9  #include "ns3/ndnSIM/model/ndn-common.hpp"
10
11 #include "ns3/nstime.h"
12 #include "ns3/ptr.h"
13
14 #include "ns3/chord-ipv4.h"
15
16 namespace ns3 {
17 namespace ndn {
18
19
20 class ChordProducer : public App {
21 public:
22   static TypeId
23   GetTypeId (void);
24
25   ChordProducer ();
26
27   // inherited from NdnApp
28   virtual void
29   OnInterest (shared_ptr<const Interest> interest);
30
31   /**
32    * \brief Respond to an interest upon data retrieved from the DHT
33    */
34   void
35   RespondWithData (shared_ptr<const Interest> interest);
36
37   /**
38    * \brief Retrieve a data item from the DHT (key = interest sequence number).
39    * Adds the interest to the ongoing DHT retrievals set.
```

```
40     * Returns immediately, if there is already an ongoing retrieval for some
   ↪  key.
41     */
42    void
43    RetrieveFromDHT (shared_ptr<const Interest> interest);
44
45    std::unordered_set <std::shared_ptr<const ndn::Interest>>::iterator
   ↪  FindPendingDHTretrieval (uint8_t* key);
46    /**
47     * \brief Callback invoked by ChordIpv4 upon retrieval success.
48     * Finds the pending DHT retrieval in the set and invokes response to the
   ↪  corresponding NDN interest and,
49     * removes the pending DHT retrieval from the set.
50     */
51    void RetrieveSuccess (uint8_t* key, uint8_t numBytes, uint8_t* object,
   ↪  uint32_t objectBytes);
52
53    /**
54     * \brief Callback invoked by ChordIpv4 upon retrieval failure.
55     * Finds and removes the pending DHT retrieval from the set, such that
   ↪  RetrieveFromDHT won't block anymore.
56     */
57    void RetrieveFailure (uint8_t* key, uint8_t keyBytes);
58
59  protected:
60    // inherited from Application base class.
61    virtual void
62    StartApplication (); // Called at time specified by Start
63
64    virtual void
65    StopApplication (); // Called at time specified by Stop
66
67  private:
68    Name m_prefix;
69    Name m_postfix;
70    uint32_t m_virtualPayloadSize;
71    Time m_freshness;
72
73    uint32_t m_signature;
74    Name m_keyLocator;
75
76    Ptr<ChordIpv4> m_chordIpv4;
77
78    std::unordered_set <shared_ptr<const Interest>> m_pendingDHTretrievals;
79  };
80
81  } // namespace ndn
82  } // namespace ns3
83
84  #endif // NDN_CHORD_PRODUCER_H
```

A.2.2 ndn-chord-producer.cpp

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

#include "ndn-chord-producer.hpp"
#include "ns3/log.h"
#include "ns3/string.h"
#include "ns3/uinteger.h"
#include "ns3/packet.h"
#include "ns3/simulator.h"
#include "model/ndn-l3-protocol.hpp"
#include "helper/ndn-fib-helper.hpp"

#include <openssl/sha.h>
#include <memory>
#include <string>

NS_LOG_COMPONENT_DEFINE("ndn.ChordProducer");

namespace ns3 {
namespace ndn {

NS_OBJECT_ENSURE_REGISTERED(ChordProducer);

TypeId
ChordProducer::GetTypeId(void)
{
  static TypeId tid =
    TypeId("ns3::ndn::ChordProducer")
      .SetGroupName("Ndn")
      .SetParent<App>()
      .AddConstructor<ChordProducer>()
      .AddAttribute("Prefix", "Prefix, for which producer has the data",
      ↪  StringValue("/"),
                    MakeNameAccessor(&ChordProducer::m_prefix),
                     ↪  MakeNameChecker())
      .AddAttribute(
        "Postfix",
        "Postfix that is added to the output data (e.g., for adding
        ↪  producer-uniqueness)",
        StringValue("/"), MakeNameAccessor(&ChordProducer::m_postfix),
        ↪  MakeNameChecker())
      .AddAttribute("PayloadSize", "Virtual payload size for Content packets",
      ↪  UintegerValue(1024),
                    MakeUintegerAccessor(&ChordProducer::m_virtualPayloadSize),
                    MakeUintegerChecker<uint32_t>())
      .AddAttribute("Freshness", "Freshness of data packets, if 0, then
      ↪  unlimited freshness",
                    TimeValue(Seconds(0)),
                     ↪  MakeTimeAccessor(&ChordProducer::m_freshness),
```

```
42                      MakeTimeChecker())
43      .AddAttribute(
44        "Signature",
45        "Fake signature, 0 valid signature (default), other values
          ↪ application-specific",
46        UintegerValue(0), MakeUintegerAccessor(&ChordProducer::m_signature),
47        MakeUintegerChecker<uint32_t>())
48      .AddAttribute("KeyLocator",
49                      "Name to be used for key locator.  If root, then key
                        ↪ locator is not used",
50                      NameValue(),
                        ↪ MakeNameAccessor(&ChordProducer::m_keyLocator),
                        ↪ MakeNameChecker())
51      .AddAttribute("ChordIpv4",
52                      "The ChordIpv4 app associated with this producer.",
53                      PointerValue (),
54                      MakePointerAccessor (&ChordProducer::m_chordIpv4),
55                      MakePointerChecker<ChordIpv4> ());
56    return tid;
57  }
58
59  ChordProducer::ChordProducer()
60  {
61    NS_LOG_FUNCTION_NOARGS();
62  }
63
64  // inherited from Application base class.
65  void
66  ChordProducer::StartApplication()
67  {
68    NS_LOG_FUNCTION_NOARGS();
69
70    m_chordIpv4->SetRetrieveSuccessCallback (MakeCallback
      ↪ (&ChordProducer::RetrieveSuccess, this));
71    m_chordIpv4->SetRetrieveFailureCallback (MakeCallback
      ↪ (&ChordProducer::RetrieveFailure, this));
72
73    App::StartApplication();
74
75    FibHelper::AddRoute(GetNode(), m_prefix, m_face, 0);
76  }
77
78  void
79  ChordProducer::StopApplication()
80  {
81    NS_LOG_FUNCTION_NOARGS();
82    App::StopApplication();
83  }
84
85  void
```

```
86  ChordProducer::OnInterest(shared_ptr<const Interest> interest)
87  {
88    App::OnInterest(interest); // tracing inside
89
90    NS_LOG_FUNCTION(this << interest);
91
92    if (!m_active)
93      return;
94
95    RetrieveFromDHT (interest);
96  }
97
98  void
99  ChordProducer::RespondWithData (shared_ptr<const Interest> interest)
100 {
101
102   Name dataName(interest->getName());
103
104   auto data = make_shared<Data>();
105   data->setName(dataName);
106   data-
      ↪  >setFreshnessPeriod(::ndn::time::milliseconds(m_freshness.GetMilliSeconds()));
107
108   data->setContent(make_shared< ::ndn::Buffer>(m_virtualPayloadSize));
109
110   Signature signature;
111   SignatureInfo signatureInfo(static_cast<
      ↪  ::ndn::tlv::SignatureTypeValue>(255));
112
113   if (m_keyLocator.size() > 0) {
114     signatureInfo.setKeyLocator(m_keyLocator);
115   }
116
117   signature.setInfo(signatureInfo);
118   signa-
      ↪  ture.setValue(::ndn::makeNonNegativeIntegerBlock(::ndn::tlv::SignatureValue,
      ↪  m_signature));
119
120   data->setSignature(signature);
121
122   NS_LOG_INFO("node(" << GetNode()->GetId() << ") responding with Data: " <<
      ↪  data->getName());
123
124   // to create real wire encoding
125   data->wireEncode();
126
127   m_transmittedDatas(data, this, m_face);
128   m_appLink->onReceiveData(*data);
129
130 }
```

```
131
132 void
133 ChordProducer::RetrieveFromDHT (shared_ptr<const Interest> interest)
134 {
135   std::string seqNo = std::to_string
      ↪  (interest->getName().at(-1).toSequenceNumber());
136
137   // Check whether there is already an ongoing DHT retrieval for this interest
138   auto ite = m_pendingDHTretrievals.begin ();
139   while (ite != m_pendingDHTretrievals.end ())
140   {
141     if (seqNo == std::to_string ((*ite)->getName ().at (-1).toSequenceNumber
        ↪  ()))
142     {
143       NS_LOG_INFO ("There is already an ongoing DHT retrieval for this
          ↪  interest: " << seqNo);
144       return;
145     }
146     ++ite;
147   }
148
149   NS_LOG_INFO ("Key getting queried from DHT: " << seqNo);
150
151   unsigned char* md = (unsigned char*) malloc (20);
152   const unsigned char* message = (const unsigned char*) seqNo.c_str();
153   SHA1 (message , seqNo.length () , md);
154
155   m_pendingDHTretrievals.insert (interest);
156
157   m_chordIpv4->Retrieve (md, 20);
158
159   free (md);
160
161 }
162
163 std::unordered_set <std::shared_ptr<const ndn::Interest>>::iterator
164 ChordProducer::FindPendingDHTretrieval (uint8_t* key)
165 {
166   auto ite = m_pendingDHTretrievals.begin ();
167   while (ite != m_pendingDHTretrievals.end ())
168   {
169     std::string seqNo = std::to_string ((*ite)->getName ().at
        ↪  (-1).toSequenceNumber ());
170     unsigned char* md = (unsigned char*) malloc (20);
171     const unsigned char* message = (const unsigned char*) seqNo.c_str ();
172     SHA1 (message, seqNo.length (), md);
173
174     if (std::memcmp (key, md, 20) == 0)
175     {
176       free (md);
```

```
177         return ite;
178       }
179
180       free (md);
181       ++ite;
182     }
183
184   return ite;
185 }
186
187 void
188 ChordProducer::RetrieveSuccess (uint8_t* key, uint8_t numBytes, uint8_t*
    ↪  object, uint32_t objectBytes)
189 {
190   NS_LOG_INFO("Retrieve Success!");
191
192   auto ite = FindPendingDHTretrieval (key);
193
194   if (ite != m_pendingDHTretrievals.end ())
195   {
196     m_pendingDHTretrievals.erase (ite);
197     RespondWithData (*ite);
198   }
199 }
200
201 void
202 ChordProducer::RetrieveFailure (uint8_t* key, uint8_t keyBytes)
203 {
204   NS_LOG_UNCOND ("Retrieve Failure Reported...");
205
206   auto ite = FindPendingDHTretrieval (key);
207
208   if (ite != m_pendingDHTretrievals.end ())
209   {
210     m_pendingDHTretrievals.erase (ite);
211   }
212 }
213
214 } // namespace ndn
215 } // namespace ns3
```

A.3 Main Simulation Scenarios

The main two simulation programs which were used to conduct experiments are attached to this section. Their corresponding documentation can be found in section 4.7.

A.3.1 ndn-zones-chord-small-world.cc

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/point-to-point-helper.h"
#include "ns3/config-store.h"
#include "ns3/ndnSIM-module.h"
#include "chord-helper.h"

#include <vector>
#include <cstdio>
#include <iostream>


using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("NdnZonesChordSmallWorld");

// ns-3-chord boilerplate
void
ChordHelper::JoinSuccess (std::string vNodeName, uint8_t* key, uint8_t
 ↪  numBytes)
{
  NS_LOG_FUNCTION_NOARGS ();
  NS_LOG_UNCOND ("VNode: " << vNodeName << " Joined successfully!");
}

void
ChordHelper::InsertSuccess (uint8_t* key, uint8_t numBytes, uint8_t* object,
 ↪  uint32_t objectBytes)
{
  NS_LOG_FUNCTION_NOARGS ();
  NS_LOG_UNCOND ("Insert Success!");
}

void
ChordHelper::InsertFailure (uint8_t* key, uint8_t numBytes, uint8_t* object,
 ↪  uint32_t objectBytes)
{
  NS_LOG_FUNCTION_NOARGS ();
  NS_LOG_UNCOND ("Insert Failure Reported...");
}
```

```
39
40 void
41 ChordHelper::InsertVNode(Ptr<ChordIpv4> chordApplication, std::string
   ↪ vNodeName)
42 {
43   NS_LOG_FUNCTION_NOARGS ();
44   unsigned char* md = (unsigned char*) malloc (20);
45   const unsigned char* message = (const unsigned char*) vNodeName.c_str ();
46   SHA1 (message , vNodeName.length () , md);
47
48   NS_LOG_UNCOND ("Scheduling Command InsertVNode...");
49   chordApplication->InsertVNode (vNodeName, md, 20);
50   free (md);
51 }
52
53 void
54 ChordHelper::VNodeFailure (std::string vNodeName, uint8_t* key, uint8_t
   ↪ numBytes)
55 {
56   NS_LOG_FUNCTION_NOARGS ();
57   NS_LOG_UNCOND ("VNode: " << vNodeName << " Failed");
58 }
59
60 void
61 ChordHelper::Insert (Ptr<ChordIpv4> chordApplication, std::string resourceName,
   ↪ std::string resourceValue)
62 {
63   NS_LOG_FUNCTION_NOARGS ();
64   NS_LOG_UNCOND ("Insert k/V: " << resourceName << ", \"" << resourceValue <<
   ↪  "\"");
65   unsigned char* md = (unsigned char*) malloc (20);
66   const unsigned char* message = (const unsigned char*) resourceName.c_str ();
67   SHA1 (message , resourceName.length () , md);
68   uint16_t payloadSize = 1500;
69   for (uint16_t i = resourceValue.length (); i < payloadSize; i++)
70   {
71       resourceValue += "v";
72   }
73   unsigned char* value = (unsigned char *)(resourceValue.c_str ());
74   chordApplication->Insert (md, 20, value, payloadSize);
75   free (md);
76 }
77
78 void
79 ChordHelper::Retrieve (Ptr<ChordIpv4> chordApplication, std::string
   ↪ resourceName)
80 {
81   NS_LOG_FUNCTION_NOARGS ();
82   unsigned char* md = (unsigned char*) malloc (20);
83   const unsigned char* message = (const unsigned char*) resourceName.c_str ();
```

```
84    SHA1 (message , resourceName.length () , md);
85    chordApplication->Retrieve (md, 20);
86    free (md);
87  }
88
89  void
90  ChordHelper::RetrieveFailure (uint8_t* key, uint8_t keyBytes)
91  {
92    NS_LOG_FUNCTION_NOARGS ();
93    NS_LOG_UNCOND ("Retrieve Failure Reported...");
94  }
95
96  void
97  ChordHelper::RetrieveSuccess (uint8_t* key, uint8_t numBytes, uint8_t* object,
    ↪  uint32_t objectBytes)
98  {
99    NS_LOG_FUNCTION_NOARGS ();
100   NS_LOG_UNCOND ("Retrieve Success!");
101 }
102
103 void
104 ChordHelper::VNodeKeyOwnership (std::string vNodeName, uint8_t* key, uint8_t
    ↪  keyBytes,
105                                 uint8_t* predecessorKey, uint8_t
                                    ↪  predecessorKeyBytes,
106                                 uint8_t* oldPredecessorKey, uint8_t
                                    ↪  oldPredecessorKeyBytes,
107                                 Ipv4Address predecessorIp, uint16_t
                                    ↪  predecessorPort)
108 {
109   NS_LOG_FUNCTION_NOARGS ();
110   NS_LOG_UNCOND ("VNode: " << vNodeName << " Key Space Ownership change
    ↪  reported");
111   NS_LOG_UNCOND ("New predecessor Ip: " << predecessorIp << " Port: " <<
    ↪  predecessorPort);
112 }
113
114 const uint16_t THROUGHPUT_COMPUTATION_PERIOD_S = 10;
115 std::map <uint16_t, double> smallWorldThroughputPerPeriod;
116
117 void
118 TraceTransitLoad (Ptr<const Packet> packet)
119 {
120   double currentTime = Simulator::Now ().GetSeconds();
121   uint16_t period = currentTime / THROUGHPUT_COMPUTATION_PERIOD_S;
122   auto pos = smallWorldThroughputPerPeriod.find (period);
123   double partialThroughputB = (double) packet->GetSize () /
    ↪  THROUGHPUT_COMPUTATION_PERIOD_S;
124   if (pos == smallWorldThroughputPerPeriod.end ())
125   {
```

```
126       smallWorldThroughputPerPeriod.insert ({period, partialThroughputB});
127     }
128     else
129     {
130       pos->second += partialThroughputB;
131     }
132   }
133
134   int
135   main (int argc, char *argv[])
136   {
137     uint16_t noUEs = 5;
138     uint16_t noDataItems = 100;
139     double interestsPerS = 1.0;
140     double simulationRuntimeM = 10;
141     uint16_t csSize = 10;
142     CommandLine cmd;
143     cmd.AddValue ("noDataItems",
144                   "Number of data items (sequence numbers) to use in the
                      ↪  simulation",
145                   noDataItems);
146     cmd.AddValue ("interestsPerS",
147                   "Frequency of sending out interest packets to be used by
                      ↪  consumer apps",
148                   interestsPerS);
149     cmd.AddValue ("noUEs",
150                   "Number of UEs per last-mile zone",
151                   noUEs);
152     cmd.AddValue ("simulationRuntimeM",
153                   "Simulation run time (after start of ndnSIM apps)",
154                   simulationRuntimeM);
155     cmd.AddValue ("csSize",
156                   "Max. Content Store size to use on NDN nodes with activated
                      ↪  CS",
157                   csSize);
158     cmd.Parse (argc, argv);
159     const uint8_t NO_ZONES = 5;
160     const uint8_t NO_SMALL_WORLD_NODES = 8;
161     NodeContainer smallWorldNodes;
162     // Edges from Watts-Strogatz random small-world network graph, with n = 8, k
      ↪  = 3, p = 0.7
163     const std::vector <std::pair <int, int>> smallWorldEdges =
164     {
165       std::make_pair (0, 3),
166       std::make_pair (1, 7),
167       std::make_pair (2, 5),
168       std::make_pair (2, 3),
169       std::make_pair (2, 6),
170       std::make_pair (3, 4),
171       std::make_pair (3, 7),
```

```
172       std::make_pair (5, 7)
173     };
174     NodeContainer superNodes;
175     ChordHelper chordHelper;
176     const uint16_t CHORD_PORT = 2000;
177     double INTER_VNODE_INSERTION_GAP_S = 70.0;
178     double INTER_CHORD_INSERTION_GAP_S = 0.5;
179     double CHORD_INSERTION_STARTTIME_S = NO_ZONES * INTER_VNODE_INSERTION_GAP_S;
180     double NDN_APPS_START_TIME_S = CHORD_INSERTION_STARTTIME_S +
        ↪  INTER_CHORD_INSERTION_GAP_S * noDataItems;
181     const std::string LTE_THROUGHPUT_5_UE = "4091.08342373628kbps";
182     const double FIBER_LATENCY_NS_M = 5.0;
183     const double BBU_RRH_DISTANCE_M = 15.0;
184     const std::vector<double> UE_DELAYS_MS = {14.937298, 14.938553, 15.937298,
        ↪  15.938553, 15.939808};
185     std::vector<NodeContainer> zoneUEs (NO_ZONES);
186     std::vector<NodeContainer> zoneENBelems (NO_ZONES);
187     const std::string NDN_PREFIX = "/payload";
188
189     // Set up small world network
190     smallWorldNodes.Create (NO_SMALL_WORLD_NODES);
191     InternetStackHelper internetStackHelper;
192     internetStackHelper.Install (smallWorldNodes);
193
194     PointToPointHelper smallWorldLink;
195     smallWorldLink.SetDeviceAttribute ("DataRate", StringValue ("100Mbps"));
196     smallWorldLink.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (10)));
197
198     // Set up small-world network edges in terms of p2p links and assigning IP
        ↪  addresses
199     Ipv4AddressHelper ipv4AddressHelper;
200     for (uint8_t i = 0; i < smallWorldEdges.size (); i++)
201     {
202       NetDeviceContainer smallWorldDevices =
203           smallWorldLink.Install (smallWorldNodes.Get (smallWorldEdges[i].first),
204                                   smallWorldNodes.Get
                                        ↪  (smallWorldEdges[i].second));
205       smallWorldDevices.Get (0)->TraceConnectWithoutContext ("PhyTxEnd",
        ↪  MakeCallback (&TraceTransitLoad));
206       smallWorldDevices.Get (1)->TraceConnectWithoutContext ("PhyTxEnd",
        ↪  MakeCallback (&TraceTransitLoad));
207       std::string base = "10.1." + std::to_string (i+1) + ".0";
208       ipv4AddressHelper.SetBase (base.c_str (), "255.255.255.0");
209       ipv4AddressHelper.Assign (smallWorldDevices);
210     }
211
212     // Automatically set up IPv4 routing among small-world nodes
213     Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
214
215     // Picking random small-world nodes as super nodes
```

```
216   superNodes.Add (smallWorldNodes.Get (0));
217   superNodes.Add (smallWorldNodes.Get (2));
218   superNodes.Add (smallWorldNodes.Get (3));
219   superNodes.Add (smallWorldNodes.Get (5));
220   superNodes.Add (smallWorldNodes.Get (7));
221   NS_ASSERT_MSG (superNodes.size () == NO_ZONES, "No. of super nodes does not
      ↪  match no. of zones!");
222
223   // Set up Chord on the super nodes
224   for (uint8_t i = 0; i < NO_ZONES; i++)
225   {
226     ChordIpv4Helper chordIpv4Helper (superNodes.Get (0)->GetObject<Ipv4>
      ↪  ()->GetAddress (1, 0).GetLocal (),
227                                      CHORD_PORT,
228                                      superNodes.Get (i)->GetObject<Ipv4>
                                       ↪  ()->GetAddress (1, 0).GetLocal (),
229                                      CHORD_PORT,
230                                      CHORD_PORT + 1,
231                                      CHORD_PORT + 2);
232     ApplicationContainer chordApps = chordIpv4Helper.Install (superNodes.Get
      ↪  (i));
233     chordApps.Start (Seconds (0.0));
234     Ptr<ChordIpv4> chordIpv4 = superNodes.Get (i)->GetApplication
      ↪  (0)->GetObject<ChordIpv4> ();
235     chordIpv4->SetJoinSuccessCallback (MakeCallback (&ChordHelper::JoinSuccess,
      ↪  &chordHelper));
236     chordIpv4->SetInsertSuccessCallback (MakeCallback
      ↪  (&ChordHelper::InsertSuccess, &chordHelper));
237     chordIpv4->SetInsertFailureCallback (MakeCallback
      ↪  (&ChordHelper::InsertFailure, &chordHelper));
238     chordIpv4->SetVNodeFailureCallback (MakeCallback
      ↪  (&ChordHelper::VNodeFailure, &chordHelper));
239     chordIpv4->SetVNodeKeyOwnershipCallback (MakeCallback
      ↪  (&ChordHelper::VNodeKeyOwnership, &chordHelper));
240
241     // Setting up the bootstrap node at the first iteration
242     if (i == 0)
243     {
244       Simulator::Schedule (Seconds (0.01), &ChordHelper::InsertVNode,
        ↪  &chordHelper, chordIpv4, "bootNode");
245     }
246     // Prepare ChordIpv4 applications on all (non-bootstrap) super nodes
247     else
248     {
249       std::string nodeName = "superNode" + std::to_string (i);
250       Simulator::Schedule (Seconds (i * INTER_VNODE_INSERTION_GAP_S),
        ↪  &ChordHelper::InsertVNode, &chordHelper, chordIpv4, nodeName);
251     }
252   }
253
```

```
254   // Insert data items into the DHT (cycling through the supernodes)
255   for (uint8_t k = 1; k <= noDataItems; k++)
256   {
257     Ptr<ChordIpv4> superNodeApp = superNodes.Get ((k-1) %
      ↪  NO_ZONES)->GetApplication (0)->GetObject<ChordIpv4> ();
258     std::string payloadName = "payload" + std::to_string (k);
259     Simulator::Schedule (Seconds (CHORD_INSERTION_STARTTIME_S + (k-1) *
      ↪  INTER_CHORD_INSERTION_GAP_S),
260                          &ChordHelper::Insert,
261                          &chordHelper,
262                          superNodeApp,
263                          std::to_string (k), payloadName);
264   }
265
266
267   // Set up last-mile LTE zones
268   PointToPointHelper enbPtp;
269   // Apply computed throughput to emulate a shared channel on BBU-RRH link,
      ↪  over which zone UEs compete for bandwidth
270   enbPtp.SetDeviceAttribute ("DataRate", DataRateValue (DataRate
      ↪  (LTE_THROUGHPUT_5_UE)));
271
272   // Assumption: Using optical fiber to interconnect RRH and BBU
273   enbPtp.SetChannelAttribute ("Delay", TimeValue (NanoSeconds
      ↪  (FIBER_LATENCY_NS_M * BBU_RRH_DISTANCE_M)));
274   for (uint16_t i = 0; i < NO_ZONES; i++)
275   {
276     NodeContainer& enbElements = zoneENBelems[i];
277     // Add this zone's BBU node and associate it with a super node (part of the
      ↪  small-world network)
278     enbElements.Add (superNodes.Get (i));
279     // Create this zone's RRH node
280     enbElements.Create (1);
281     enbPtp.Install (enbElements);
282
283     // Create UE nodes
284     NodeContainer& ueNodes = zoneUEs[i];
285     ueNodes.Create (noUEs);
286
287     // Set up p2p link among each zone UE and the RRH
288     for (uint8_t i = 0; i < noUEs; i++)
289     {
290       PointToPointHelper ueRrhPtpLink;
291       // Applying avg. measured UE delays
292       ueRrhPtpLink.SetChannelAttribute ("Delay", TimeValue (MilliSeconds
        ↪  (UE_DELAYS_MS[i % UE_DELAYS_MS.size ()])));
293       // Setting a high data rate to "overwrite" the very low default on
        ↪  PointToPointNetDevice
294       ueRrhPtpLink.SetDeviceAttribute ("DataRate", DataRateValue (DataRate
        ↪  ("1Gbps")));
```

```
295         ueRrhPtpLink.Install (enbElements.Get (1), ueNodes.Get (i));
296       }
297     }
298
299     // Set up NDN in the last-mile zones
300     ns3::ndn::StackHelper ndnHelper;
301     // Use Least recently used (LRU) as the Content Store strategy (discards the
        ↪  least recently used items first)
302     ndnHelper.setPolicy("nfd::cs::lru");
303     ndnHelper.SetDefaultRoutes (true);
304
305     // Disable the Content Store (CS) on all the UE nodes (1 is the minimum)
306     ndnHelper.setCsSize (1);
307     for (uint8_t i = 0; i < NO_ZONES; i++)
308     {
309       ndnHelper.Install (zoneUEs[i]);
310     }
311     // Set default CS size (in no. of packets) on all other nodes
312     ndnHelper.setCsSize (csSize);
313     ndnHelper.Install (smallWorldNodes);
314     for (uint8_t i = 0; i < NO_ZONES; i++)
315     {
316       ndnHelper.Install (zoneENBelems[i].Get (1));
317     }
318
319     // Set BestRoute strategy
320     ns3::ndn::StrategyChoiceHelper::InstallAll (NDN_PREFIX,
        ↪  "/localhost/nfd/strategy/best-route");
321     ns3::ndn::GlobalRoutingHelper ndnGlobalRoutingHelper;
322     // Installing global routing interface on all nodes
323     ndnGlobalRoutingHelper.InstallAll ();
324
325     // Set up producer application for BBU nodes
326     ns3::ndn::AppHelper producerHelper ("ns3::ndn::ChordProducer");
327     producerHelper.SetPrefix (NDN_PREFIX);
328     producerHelper.SetAttribute ("PayloadSize", StringValue ("1500"));
329
330     // Set up consumer applications for UE nodes
331     ns3::ndn::AppHelper consumerHelper ("ns3::ndn::ConsumerZipfMandelbrot");
332     consumerHelper.SetPrefix (NDN_PREFIX);
333     consumerHelper.SetAttribute ("Frequency", StringValue (std::to_string
        ↪  (interestsPerS)));
334     consumerHelper.SetAttribute ("Randomize", StringValue ("uniform"));
335     // The higher s (parameter of power), the more popular items of small
        ↪  sequence numbers (^= rank)
336     consumerHelper.SetAttribute ("s", StringValue("1.2"));
337     consumerHelper.SetAttribute ("NumberOfContents", StringValue (std::to_string
        ↪  (noDataItems)));
338
339     // Install producer and consumer apps in each last-mile zone
```

```
340   for (uint8_t i = 0; i < NO_ZONES; i++)
341   {
342     Ptr<Node> producer = superNodes.Get (i);
343     Ptr<ChordIpv4> superNodeChord = producer->GetApplication (0)->GetObject
      ↪  <ChordIpv4> ();
344     producerHelper.SetAttribute ("ChordIpv4", PointerValue (superNodeChord));
345     ndnGlobalRoutingHelper.AddOrigin (NDN_PREFIX, producer);
346     ApplicationContainer producerApp = producerHelper.Install (producer);
347     producerApp.Start (Seconds (NDN_APPS_START_TIME_S));
348
349     for (uint8_t n = 0; n < noUEs; n++)
350     {
351       ApplicationContainer consumerApp = consumerHelper.Install (zoneUEs[i].Get
        ↪  (n));
352       consumerApp.Start (Seconds (NDN_APPS_START_TIME_S));
353     }
354   }
355
356   ns3::ndn::GlobalRoutingHelper::CalculateRoutes ();
357
358   // Do the actual simulation
359   Simulator::Stop (Seconds (NDN_APPS_START_TIME_S) + Minutes
      ↪  (simulationRuntimeM));
360
361   std::string params =  std::to_string (noUEs) + "_" +
362                         std::to_string ((uint8_t) interestsPerS) + "_" +
363                         std::to_string (noDataItems) + "_" +
364                         std::to_string (ns3::RngSeedManager::GetSeed ());
365
366   // Collect and export application-level delay stats
367   ns3::ndn::AppDelayTracer::InstallAll ("traces/small-world-chord-app-delays_"
      ↪  + params + ".tsv");
368
369   Simulator::Run ();
370   Simulator::Destroy ();
371
372   // Writing periodic backbone throughput to file
373   std::ofstream outfile;
374   outfile.open ("traces/small-world-chord-backbone-throughput_" + params +
      ↪  ".csv");
375   outfile << "periodS,throughputBps" << std::endl;
376   for (auto ite = smallWorldThroughputPerPeriod.begin (); ite !=
      ↪  smallWorldThroughputPerPeriod.end (); ite++)
377   {
378     outfile << ite->first * THROUGHPUT_COMPUTATION_PERIOD_S << "," <<
        ↪  ite->second << std::endl;
379   }
380   outfile.close ();
381   return 0;
382 }
```

A.3.2 ndn-zones-buckets-small-world.cc

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/point-to-point-helper.h"
#include "ns3/config-store.h"
#include "ns3/ndnSIM-module.h"

#include <vector>
#include <cstdio>
#include <iostream>


using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("NdnZonesBucketsSmallWorld");

const uint16_t THROUGHPUT_COMPUTATION_PERIOD_S = 10;
std::map <uint16_t, double> smallWorldThroughputPerPeriod;

void
TraceTransitLoad (Ptr<const Packet> packet)
{
  double currentTime = Simulator::Now ().GetSeconds();
  uint16_t period = currentTime / THROUGHPUT_COMPUTATION_PERIOD_S;
  auto pos = smallWorldThroughputPerPeriod.find (period);
  double partialThroughputB = (double) packet->GetSize () /
  ↪  THROUGHPUT_COMPUTATION_PERIOD_S;
  if (pos == smallWorldThroughputPerPeriod.end ())
  {
    smallWorldThroughputPerPeriod.insert ({period, partialThroughputB});
  }
  else
  {
    pos->second += partialThroughputB;
  }
}

int
main (int argc, char *argv[])
{
  uint16_t noUEs = 5;
  uint16_t noDataItems = 100;
  double interestsPerS = 1.0;
  double simulationRuntimeM = 10;
  uint16_t csSize = 10;
  CommandLine cmd;
  cmd.AddValue ("noDataItems",
```

```
47                  "Number of data items (sequence numbers) to use in the
                    ↪  simulation",
48                  noDataItems);
49    cmd.AddValue ("interestsPerS",
50                  "Frequency of sending out interest packets to be used by
                    ↪  consumer apps",
51                  interestsPerS);
52    cmd.AddValue ("noUEs",
53                  "Number of UEs per last-mile zone",
54                  noUEs);
55    cmd.AddValue ("simulationRuntimeM",
56                  "Simulation run time (after start of ndnSIM apps)",
57                  simulationRuntimeM);
58    cmd.AddValue ("csSize",
59                  "Max. Content Store size to use on NDN nodes with activated
                    ↪  CS",
60                  csSize);
61    cmd.Parse (argc, argv);
62    const uint8_t NO_ZONES = 5;
63    const uint8_t NO_SMALL_WORLD_NODES = 8;
64    NodeContainer smallWorldNodes;
65    // Edges from Watts-Strogatz random small-world network graph, with n = 8, k
      ↪  = 3, p = 0.7
66    const std::vector <std::pair <int, int>> smallWorldEdges =
67    {
68      std::make_pair (0, 3),
69      std::make_pair (1, 7),
70      std::make_pair (2, 5),
71      std::make_pair (2, 3),
72      std::make_pair (2, 6),
73      std::make_pair (3, 4),
74      std::make_pair (3, 7),
75      std::make_pair (5, 7)
76    };
77    NodeContainer superNodes;
78    const std::string LTE_THROUGHPUT_5_UE = "4091.08342373628kbps";
79    const double FIBER_LATENCY_NS_M = 5.0;
80    const double BBU_RRH_DISTANCE_M = 15.0;
81    const std::vector<double> UE_DELAYS_MS = {14.937298, 14.938553, 15.937298,
      ↪  15.938553, 15.939808};
82    std::vector<NodeContainer> zoneUEs (NO_ZONES);
83    std::vector<NodeContainer> zoneENBelems (NO_ZONES);
84    const std::string NDN_PREFIX = "/producer";
85
86    // Set up small world network
87    smallWorldNodes.Create (NO_SMALL_WORLD_NODES);
88
89    PointToPointHelper smallWorldLink;
90    smallWorldLink.SetDeviceAttribute ("DataRate", StringValue ("100Mbps"));
91    smallWorldLink.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (10)));
```

```
92
93    // Set up small-world network edges in terms of p2p links
94    for (uint8_t i = 0; i < smallWorldEdges.size (); i++)
95    {
96      NetDeviceContainer smallWorldDevices =
97          smallWorldLink.Install (smallWorldNodes.Get (smallWorldEdges[i].first),
98                                  smallWorldNodes.Get
                                    ↪  (smallWorldEdges[i].second));
99      smallWorldDevices.Get (0)->TraceConnectWithoutContext ("PhyTxEnd",
        ↪  MakeCallback (&TraceTransitLoad));
100     smallWorldDevices.Get (1)->TraceConnectWithoutContext ("PhyTxEnd",
        ↪  MakeCallback (&TraceTransitLoad));
101   }
102
103   // Picking random small-world nodes as super nodes
104   superNodes.Add (smallWorldNodes.Get (0));
105   superNodes.Add (smallWorldNodes.Get (2));
106   superNodes.Add (smallWorldNodes.Get (3));
107   superNodes.Add (smallWorldNodes.Get (5));
108   superNodes.Add (smallWorldNodes.Get (7));
109   NS_ASSERT_MSG (superNodes.size () == NO_ZONES, "No. of super nodes does not
      ↪  match no. of zones!");
110
111
112   // Set up last-mile LTE zones
113   PointToPointHelper enbPtp;
114   // Apply computed throughput to emulate a shared channel on BBU-RRH link,
      ↪  over which zone UEs compete for bandwidth
115   enbPtp.SetDeviceAttribute ("DataRate", DataRateValue (DataRate
      ↪  (LTE_THROUGHPUT_5_UE)));
116
117   // Assumption: Using optical fiber to interconnect RRH and BBU
118   enbPtp.SetChannelAttribute ("Delay", TimeValue (NanoSeconds
      ↪  (FIBER_LATENCY_NS_M * BBU_RRH_DISTANCE_M)));
119   for (uint16_t i = 0; i < NO_ZONES; i++)
120   {
121     NodeContainer& enbElements = zoneENBelems[i];
122     // Add this zone's BBU node and associate it with a super node (part of the
        ↪  small-world network)
123     enbElements.Add (superNodes.Get (i));
124     // Create this zone's RRH node
125     enbElements.Create (1);
126     enbPtp.Install (enbElements);
127
128     // Create UE nodes
129     NodeContainer& ueNodes = zoneUEs[i];
130     ueNodes.Create (noUEs);
131
132     // Set up p2p link among each zone UE and the RRH
133     for (uint8_t i = 0; i < noUEs; i++)
```

```
134       {
135         PointToPointHelper ueRrhPtpLink;
136         // Applying avg. measured UE delays
137         ueRrhPtpLink.SetChannelAttribute ("Delay", TimeValue (MilliSeconds
            ↪  (UE_DELAYS_MS[i % UE_DELAYS_MS.size ()])));
138         // Setting a high data rate to "overwrite" the very low default on
            ↪  PointToPointNetDevice
139         ueRrhPtpLink.SetDeviceAttribute ("DataRate", DataRateValue (DataRate
            ↪  ("1Gbps")));
140         ueRrhPtpLink.Install (enbElements.Get (1), ueNodes.Get (i));
141       }
142     }
143
144     // Set up NDN in the last-mile zones
145     ns3::ndn::StackHelper ndnHelper;
146     // Use Least recently used (LRU) as the Content Store strategy (discards the
        ↪  least recently used items first)
147     ndnHelper.setPolicy("nfd::cs::lru");
148     ndnHelper.SetDefaultRoutes (true);
149
150     // Disable the Content Store (CS) on all the UE nodes (1 is the minimum)
151     ndnHelper.setCsSize (1);
152     for (uint8_t i = 0; i < NO_ZONES; i++)
153     {
154       ndnHelper.Install (zoneUEs[i]);
155     }
156     // Set default CS size (in no. of packets) on all other nodes
157     ndnHelper.setCsSize (csSize);
158     ndnHelper.Install (smallWorldNodes);
159     for (uint8_t i = 0; i < NO_ZONES; i++)
160     {
161       ndnHelper.Install (zoneENBelems[i].Get (1));
162     }
163
164     // Set BestRoute strategy
165     ns3::ndn::StrategyChoiceHelper::InstallAll (NDN_PREFIX,
        ↪  "/localhost/nfd/strategy/best-route");
166     ns3::ndn::GlobalRoutingHelper ndnGlobalRoutingHelper;
167     // Installing global routing interface on all nodes
168     ndnGlobalRoutingHelper.InstallAll ();
169
170     // Set up producer application for BBU nodes
171     ns3::ndn::AppHelper producerHelper ("ns3::ndn::Producer");
172     producerHelper.SetAttribute ("PayloadSize", StringValue ("1500"));
173
174     // Set up consumer applications for UE nodes
175     ns3::ndn::AppHelper consumerHelper ("ns3::ndn::ConsumerZipfBuckets");
176     consumerHelper.SetPrefix (NDN_PREFIX);
177     consumerHelper.SetAttribute ("Frequency", StringValue (std::to_string
        ↪  (interestsPerS)));
```

```
178   consumerHelper.SetAttribute ("NumberOfPrefixBuckets", StringValue
      ↪  (std::to_string (NO_ZONES)));
179   consumerHelper.SetAttribute ("Randomize", StringValue ("uniform"));
180   // The higher s (parameter of power), the more popular items of small
      ↪  sequence numbers (^= rank)
181   consumerHelper.SetAttribute ("s", StringValue("1.2"));
182   consumerHelper.SetAttribute ("NumberOfContents", StringValue (std::to_string
      ↪  (noDataItems)));
183
184   // Install producer and consumer apps in each last-mile zone
185   for (uint8_t i = 0; i < NO_ZONES; i++)
186   {
187     Ptr<Node> producer = superNodes.Get (i);
188     std::string composedPrefix = NDN_PREFIX + std::to_string (i);
189     producerHelper.SetPrefix (composedPrefix);
190     ndnGlobalRoutingHelper.AddOrigin (composedPrefix, producer);
191     ApplicationContainer producerApp = producerHelper.Install (producer);
192
193     for (uint8_t n = 0; n < noUEs; n++)
194     {
195       ApplicationContainer consumerApp = consumerHelper.Install (zoneUEs[i].Get
          ↪  (n));
196     }
197   }
198
199   ns3::ndn::GlobalRoutingHelper::CalculateRoutes ();
200
201   // Do the actual simulation
202   Simulator::Stop (Minutes (simulationRuntimeM));
203
204   std::string params =  std::to_string (noUEs) + "_" +
205                         std::to_string ((uint8_t) interestsPerS) + "_" +
206                         std::to_string (noDataItems) + "_" +
207                         std::to_string (ns3::RngSeedManager::GetSeed ());
208
209   // Collect and export application-level delay stats
210   ns3::ndn::AppDelayTracer::InstallAll ("traces/small-world-ndn-app-delays_" +
      ↪  params + ".tsv");
211
212   Simulator::Run ();
213   Simulator::Destroy ();
214
215   // Writing periodic backbone throughput to file
216   std::ofstream outfile;
217
218   outfile.open ("traces/small-world-ndn-backbone-throughput_" + params +
      ↪  ".csv");
219   outfile << "periodS,throughputBps" << std::endl;
220
```

```
221   for (auto ite = smallWorldThroughputPerPeriod.begin (); ite !=
      ↪  smallWorldThroughputPerPeriod.end (); ite++)
222   {
223     outfile << ite->first * THROUGHPUT_COMPUTATION_PERIOD_S << "," <<
        ↪  ite->second << std::endl;
224   }
225
226   outfile.close ();
227
228   return 0;
229
230 }
```

Appendix B

Installation & Execution Guidelines

B.1 Setup

1. Extract file `source_code.zip` (delivered separately) on a Unix-based OS and navigate into subdirectory `ns3-ndn-chord-lte`.

2. Enter the following command to run the configuration process of the `Waf` build automation tool:
   `$ ./waf configure --enable-tests --enable-examples`

3. If the logged output ends with "`'configure' finished successfully`", continue with step 4. Otherwise, install missing dependencies (indicated in red) and then repeat from step 2.

4. Enter the following command to compile the NS-3 source code:
   `$ ./waf build`

B.2 Running Simulations

B.2.1 NDN & Chord Architecture

To conduct simulation experiments with the NDN & Chord architecture, enter the following command (replace X with the no. of UEs per last-mile zone and Y with the frequency of issued interests per second):
`$ ./waf --run "ndn-zones-chord-small-world --noUEs=X --interestsPerS=Y"`

Please note that the default small-world transit network is of size N=8. To use the larger N=32 transit network instead, please comment l. 190 – 200 & l. 279 – 284 and uncomment l. 202 – 236 & 287 – 292 in file `scratch/ndn-zones-chord-small-world.cc`. Furthermore, change the assigned integer on l. 187 from 8 to 32.

To retrieve logging output during the simulation, please prepend the following variable to the above command:
`NS_LOG=ndn.Consumer:ndn.ConsumerZipfMandelbrot:ndn.ChordProducer`

B.2.2 Plain NDN Reference Architecture

To conduct simulation experiments with the plain NDN reference architecture, enter the following command (replace X with the no. of UEs per last-mile zone and Y with the frequency of issued interests per second):
`$ ./waf --run "ndn-zones-buckets-small-world --noUEs=X --interestsPerS=Y"`

Please note that the default small-world transit network is of size N=8. To use the larger N=32 transit network instead, please comment l. 90 – 100 & l. 164 – 169 and un-comment l. 102 – 136 & l. 172 – 177 in file `scratch/ndn-zones-buckets-small-world.cc` (*resp.*, `scratch/ndn-zones-buckets-small-world-flooding.cc`, see below). Furthermore, change the assigned integer on l. 87 from 8 to 32.

The default NDN forwarding strategy is *Best Route*. To enforce interest *Flooding* instead, please replace the script name in the run command as follows:
`$ ./waf --run "ndn-zones-buckets-small-world-flooding [...]"`

To retrieve logging output during the simulation, please prepend the following variable to the above command(s):
`NS_LOG=ndn.Consumer:ndn.ConsumerZipfBuckets:ndn.Producer`

B.3 Metrics Collection

For both architectures, the simulation scenarios create comma- *resp.*, tab-separated result files under the directory `traces/`. The file nomenclature is the following (X = no. of UEs per domain, Y = interest frequency):

NDN & Chord:

- Retrieval delays: `small-world-chord-app-delays_X_Y_100_1.tsv`
- Transit network throughput: `small-world-chord-backbone-throughput_X_Y_100_1.csv`

Plain NDN:

- Retrieval delays: `small-world-ndn-app-delays_X_Y_100_1.tsv`
- Transit network throughput: `small-world-ndn-backbone-throughput_X_Y_100_1.csv`

Appendix C

Attached Contents

- `Abstract.txt, Zusfsg.txt`: Abstract text files in English and German
- `Masterarbeit.pdf`: This thesis document in the PDF format
- `Masterarbeit.ps`: This thesis document in the PostScript format
- `MScThesis.zip`: The compressed LaTeX project source of this thesis document
- `MidtermPresentation.pdf`: The presentation slides of the midterm presentation
- * `results.zip`: The compressed retrieval delay and transit network throughput result files of several conducted simulation experiments (*cf.* section B.3 for file nomenclature)
- * `source_code.zip`: The compressed source code for running the developed simulation programs

* directly delivered to Eryk Schiller, due to large file size