Implementation of Session Support in ndnSIM

Bachelor Project

By
Guillaume Corsini

10-931-749
guillaume.corsini@students.unibe.ch

Professor: Prof. Dr. Torsten Braun
Supervisor: Mikael Gasparyan

Communication and Distributed Systems group
Institute of Computer Science (INF)
University of Bern, Switzerland

Table of Contents

List of Figures

Glossary

CCN	Content-Centric Networking
CS	Content Store
FIB	Forwarding Interest Base
NDN	Name Domain Networking
NFD	NDN Forwarding Daemon
PIT	Pending Interest Table
SCN	Service-Centric Networking

1 Introduction

1.1 Motivation

The current Internet architecture is based on host-to-host connections, wherefore any requests have to be forwarded to the correct host. Future Internet paradigms propose architectures that focus on the data or information instead of the host. One of those paradigms is called Content-Centric Networking (CCN), in which the content is addressed directly by its given name. This paradigm has been implemented into CCNx. Another concept based on CCN is called Named Data Networking (NDN), which was originally branched from the CCNx code and further developed.

As the use of Internet today is usually not only based on content, but also often relies on services, researchers have proposed further developments of the Future Internet. The most prominent one is the service oriented architecture called Service-Centric Networking (SCN).

In order to use services, Sessions are essential in order to allow sending several messages between two communicating parties. In this Bachelor Project, a Service Session Support concept is implemented in the simulator ndnSIM. This is a new simulator that has been developed in recent years and integrates NDN into the popular network simulator ns-3 simplifying the implementation of simulations.

1.2 Task Formulation

Work on this Project consisted of the following tasks:

- Getting familiar with theoretical concepts and the ndnSIM Simulator
  The goal of this task is to understand the concepts of the Future Internet Architectures CCN/NDN, SCN and Session Support. Additionally, the basic design of the ndnSIM Simulator should be understood.
- Implementing the Session Support concept
  Propose a design for a Session Support concept in NDN and implement it ndnSIM.
- Evaluating the implemented Session Support concept in ndnSIM
  The implementation should be compared to any other existing solution.

1.3 Outline

This first Chapter provided the motivation for this Bachelor Project, its task formulation as well as this outline.

Chapter 2 will elaborate on the theoretical aspects needed for the implementation. It is separated into three sections: Future Internet, Session Support concept, and ndnSIM. In the first part three new Networking Architectures, namely Content-Centric Networking, Service-Centric Networking and Name Domain Networking, are explained. In the second section the Session Support concept is introduced. Finally, the last part presents the simulator ndnSIM in which the Session Support will be developped.

At the beginning of chapter 3, the design implemented in ndnSIM is described. The next three sections focus on the implementation by explaining several code snippets and as well as the adaptions required in the code to support multiple Consumers.

The fourth chapter shows in an evaluation, that the implemented design reduces the load on Producers compared to the current strategies.

Finally, the last chapter concludes this Bachelor Project and presents future work.

2 Related Works

2.1 Future Internet Paradigms

2.1.1 Content-Centric Networking (CCN) and Named Data Networking (NDN)

Content-Centric Networking (CCN) and Named Data Networking (NDN) are new Networking Architectures which intend to make content directly addressable and routable [1]. They are almost identical, but while CCN is developed solely by PARC, while NDN is an NSF-funded project [2]. CCN/NDN are Information-centric Networks and hence enable the exchange of content request messages ("Interests") and the content return message ("Content Objects") [3]. The goal of the CCN/NDN architecture is to address the Internet's modern-day requirements by providing a more secure, flexible and scalable network [4]:

- Security is addressed by securing the data itself and not the communication pipe in a host-to-host connection.
- Flexibility is provided in the communication by using names. These names in a CCN network are very adaptable and can be located dependently or independently. Therefore, any element in the network can make an advanced choice based on the Interest and Content Objects.
- Scalability is enabled in CCN by allowing caching and facilitating resource planning. The advantage of resource planning can be achieved through native multicast traffic and native load balancing.

CCN/NDN use the same shape of an hourglass like the current Internet for its architecture [5]. Currently, the thin waist is the universal network layer (IP), which implements a minimal functionality without unnecessary constraints for innovations. In CCN/NDN the narrow waist will be replaced by application names as seen in Figure 2.1-1.

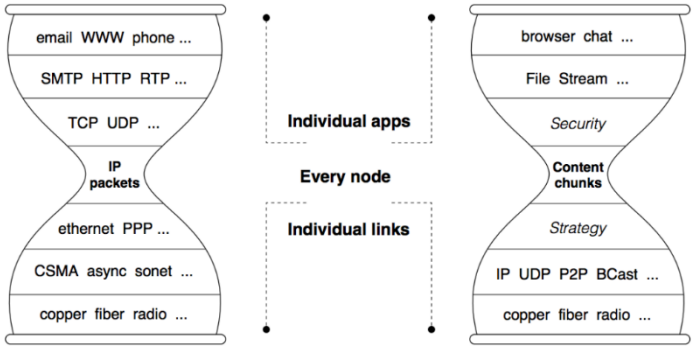

Figure 2.1-1 Narrow waist of current Internet (left) and narrow waist of CCN/NDN (right) [6].

To ensure that the current Internet continues to function on CCN/NDN, it was designed as a "universal overlay". This is clearly stated in regard to NDN in [6]: "NDN can run over anything […] and anything can run over NDN, including IP."

The CCN/NDN architecture requires two types of packets: "Interest packet" and "Data packet" [7].

- "Interest packet" contain the Content Name, a Selector (order preference, publisher filter, scope, etc.) and Nonce. They are also often called "Interest message" or simply "Interest".
- "Data packet" contain the Content Name, its signature (digest algorithm, witness, etc.), the signed information regarding the signature (publisher ID, key locator, stale time, etc.) and Data. These packets are sometimes also referred to as "Content Object".

The Content Name is a hierarchical series of words separated by a slash (similar to path names in UNIX systems) that are assigned by the Content publisher. The Selector is an optional element, that qualifies the Data matching the Interest further [8]. An example for a Selector is the Scope, which allows additional restrictions to the Content Name. Nonce are generally used to detect looping Interests [8]. The Signature is a cryptographic binding between the Content Name, the signed Information and the data, which enhances the security by providing its provenance.

For the communication in CCN/NDN, a specific router architecture is defined, which consists of three data structures and a forwarding principle. The three data structures each have their specific task [9]:

- The Forwarding Information Base (FIB) contains forwarding addresses for Interests in case the data is not cached in the rooters Content Store.
- The Pending Interest Table (PIT) stores all the Interests that are awaiting a returned Data packet including the faces that this interest came from and the one that it was sent to.
- The Content Store (CS) caches the received data allowing future requests to be satisfied directly with the data in the CS.

The forwarding principle is a strategy based on a series of policies and rules which decide how to handle any incoming Interests [9]. When a router receives an Interest, it will retrieve the longest-prefix matched entry from the FIB. Then it will decide when and where to forward this Interest. This means that the strategy can also decide to drop an Interest in a given situation, for example, if the Interest is suspected to be part of a Denial of Service attack.

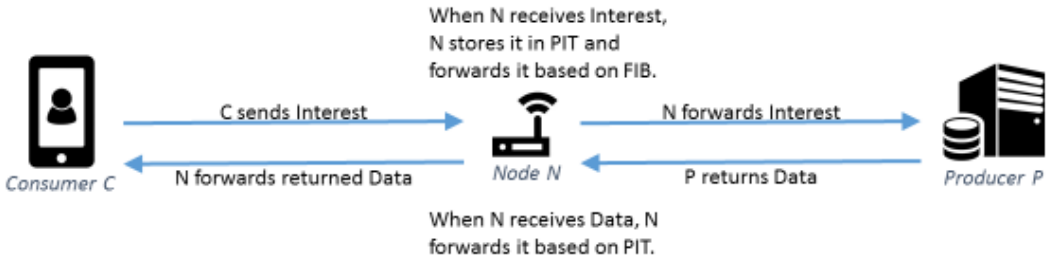

Figure 2.1-2 Example illustrating basic workflow of CCN/NDN architecture.

The following example explains the basic workflow of the CCN/NDN architecture based on the topology given in Figure 2.1-2:

When a consumer requests data in CCN/NDN, it sends an Interest message containing the name of the requested data. Each node in the network forwards the message based on its name and stores the incoming request in its Pending Interest Table (PIT). The PIT stores every Interest that awaits a returned Data Packet including its incoming and outgoing Faces. As soon as the Interest is matched to a Content Object, the "Content Object" message is returned to the consumer following the reverse path of the Interest request. This is achieved by checking the PIT states created by the Interest. It is important to note that by using caching, any "Content Object" matching the interest can be returned to the consumer. The Interest does not have to be forwarded to the content publisher [3].

The current software release of CCN is currently CCNx 1.0 and is available since November 2015 [10]. The NDN implementation was forked of the original CCNx 0.x release and has been developed separately since 2013. The current version of NDN made available in November 2016 is 0.5.0 [11].

2.1.2 Service-Centric Networking (SCN)

Service-Centric Networking (SCN) enhances the approach given by Content-Centric Network by supporting general services [12]. Service is defined by the Organization for the Advancement of Structured Information Standards (OASIS) as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistently with constraints and policies as specified by the service description" [13, p. 12].

While SCN keeps the core workflow as defined in CCN, it also adds explicit addressing for services and content processed by services.

There are multiple advantages of SCN over traditional services [12]:

- Lower delays and simplified registry for services are available because neither a lookup nor an additional registry component are required.
- Enable caching of service data by leveraging the caching features of CCN.

- Location-based services can easily be built in SCN.
- The service selection can be optimized by routing the request to the most appropriate location.

2.2 Session Support concept inspired by SOFIA

A session is a semi-permanent information interchange between two or more communicating parties [14]. These parties can be devices as well as users. The session is set up at a certain point and torn down later on. A session allows the communicating parties to transfer several messages in each direction as shown in Figure 2.2-1. Regarding the new networking architectures, those messages can be interests or data packets. These Sessions are required in order to use certain Services more efficiently. For example when a user sends a big file to a server to be sent back in a more compressed file type and the result is not as desired, the user can simply request a different file type without sending the big file again. Another example is online banking, where the user needs to be sure to stay connected to the correct server of the bank.

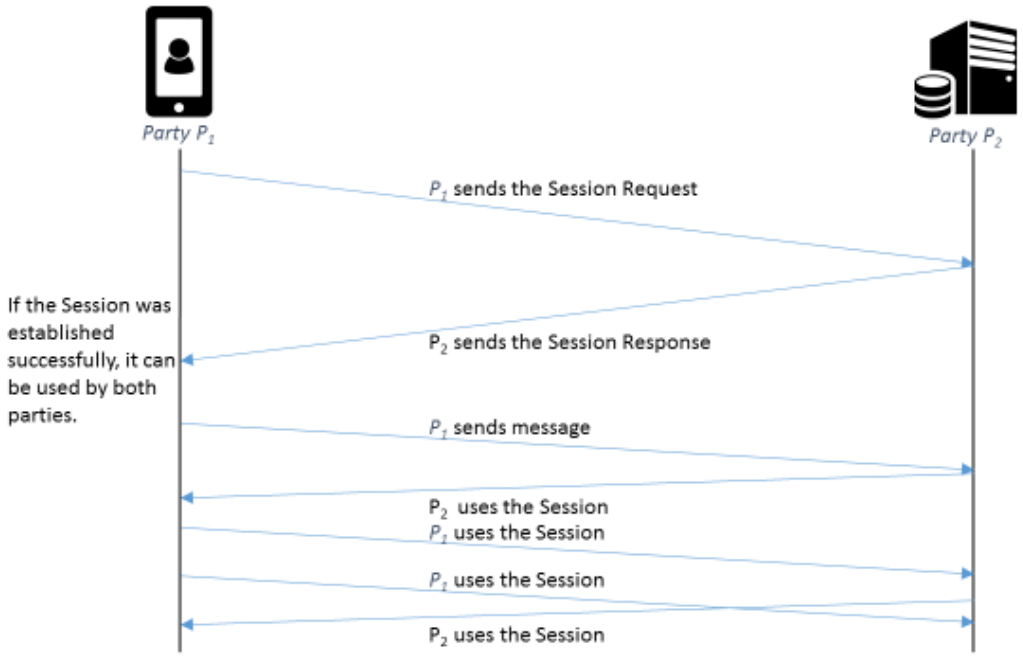

Figure 2.2-1 Workflow of a Session in general.

Services often require sessions since they consist of many messages sent between the Service consumer and the Service provider. By implementing a Session Support concept in SCN multiple possibilities for applications arise.

In the SOFIA paper [15] the architecture design is based on host and content abstraction and supports various applications beyond content retrieval. It uses two stack layers in order to increase its efficiency. The Network Layer increases the throughput of data transmissions (e.g., multipath data

transmission) and the Service Layer processes services more flexibly (e.g., service relay and service multicast). The Session Support concept proposed in SOFIA depends on both layers: While the Service Layer is used to set up a Session, using the Session relies on the Network Layer. Only the Service Layer uses service abstraction in order for the connection to be independent of the server hosting the service.

In SOFIA establishing a Service Session is a four step process:

1. The consumer *C* sends an Interest containing a generated virtual service name V_c and the requested service *S* to the network.
2. The interest is forwarded to the producer by the forwarding strategy implemented on the routers.
3. The producer *P* generates a unique service name SV_c and adds the mapping of SV_c to $<V_c , C>$ for further packet demultiplexing. *P* then replies with all available information.
4. The router forwards the response from the interest to the consumer *C* while leaving the information untouched.

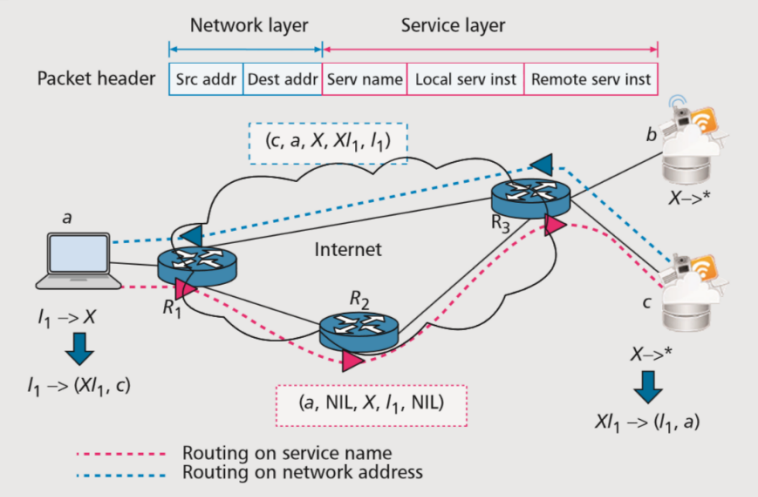

Figure 2.2-2 Establishing a Service Session as proposed by SOFIA [15].

After a Service Session has been set up, it can be used by the consumer to send further packets, which will be forwarded based on their address. The forwarding aspect goes against the core concept of NDN, which intends to enable End-to-End connections based on Names instead of Host-to-Host connections. Therefore, the Session Support concept proposed by SOFIA will be altered for the implementation in ndnSIM in order to exclude source and destination addresses.

2.3 ndnSIM Simulator

The simulator ndnSIM is based on the network simulator ns-3, wherefore most of the code is written in C++ [16]. There are already many elements proposed by ns-3, which can also be used for simulation inside ndnSIM: Although a new network-layer protocol model was used to implement ndnSIM, it can run on top of any link-layer protocol model. The current implementation of the simulator is version 2.1 [16].

In this last version the packet format fulfills the NDN packet specifications given by the Named Data Networking Consortium [16]. NdnSIM relies on the C++ library ndn-cxx, which implements NDN primitives that help implement NDN applications [17]. Finally, the simulator used the source code from NDN Forwarding Daemon (NFD) to implement NDN forwarding and management.

There are several forwarding strategies available through NFD in the simulator ndnSIM [18]. These strategies define how Interests should be handled: They can be either forwarded or alternatively dropped. By default the *Best Route Strategy* is active, which forwards Interests to the upstream that has the lowest routing cost. The *Multicast Strategy* forwards the Interest to all upstream nodes. There is also a strategy called *Client Control Strategy*, which allows a local consumer application to choose the upstream for the forwarding of each Interest. Finally, it is also possible to implement a new strategy or override an existing one if desired. The *Random Strategy* is provided as an example for a new strategy and it implements random load balancing.

These integrations enhance the simulations to be maximally realistic and enable them to be reproduced in real environments with almost no changes in the source code [16]. This means especially that experiments in ndnSIM regarding NDN forwarding can also be used with the real NFD implementation due to its very tight integration.

The documentation consists of several example files and some explanations that have been added as comments. These allow a quick overview of the simulators' capabilities and enable users to put simulations together quickly. Unfortunately, the documentation of the source code itself is rather shallow, which hinders a speedy understanding of how to modify the source code of ndnSIM.

The components and the communication layer abstraction in the simulator are shown in Figure 2.3-1. On the left, the relations between Components implemented in ndnSIM are shown. The center is the NDN protocol stack which relates to the AppFace, the NetDeviceFace, the CS and the NFD. While the AppFace connects to Applications, the NFD relates to the following five components: Face, CS, PIT, FIB and Forwarding Strategy. Finally, the NetDeviceFace relates to the NetDevices. The figure on the right shows the abstraction of the Communication Layers in ndnSIM. The NDN Protocol Stack

(L3Protocol) implements the Link Layer on the lowest level. Just above is the Network Layer and finally the UDP/TCP protocols are contained.

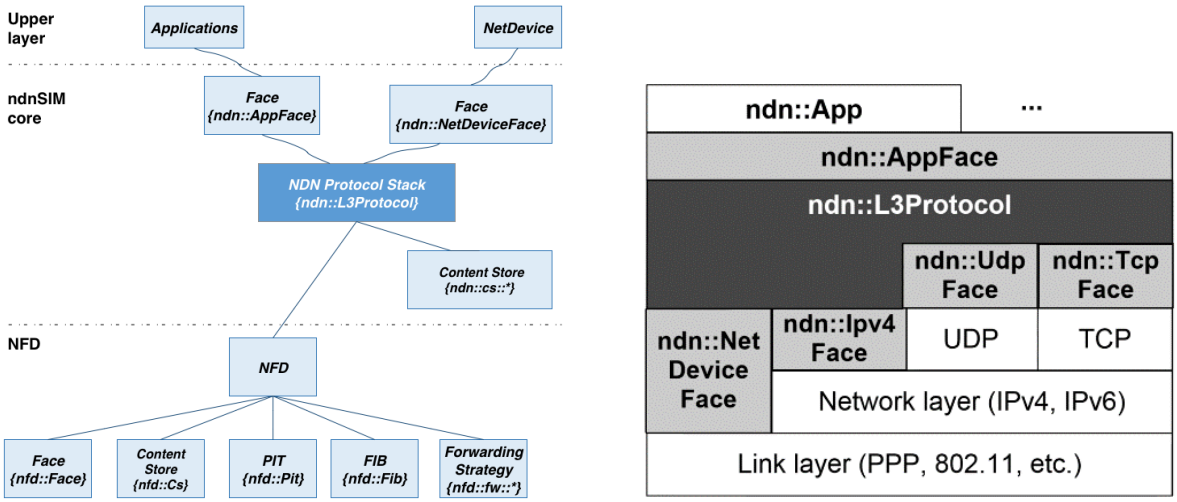

Figure 2.3-1 Components (left) and Communication Layer abstraction (right) of the ndnSIM simulator.

3 Design and Implementation

3.1 Design

The Session Support concept from the paper SOFIA [15] uses addresses in the Network Layer in order to increase the throughput of data transmissions as stated in Chapter 2.2. In contrast, ndnSIM having implemented NDN intends to refrain from using them. Therefore, the Session Support concept that will be implemented is inspired by the proposition made in the SOFIA paper, but removes the use of addresses.

The idea is that a Consumer C, who requests a Service, and Producer P, which provides this service, both generate a unique identifier each. By combining both unique identifier a session identifier is defined. This session identifier is then used to add a new route into the FIB which then knows how to forward Interests that contain the session identifier.

For example if a Consumer C would like to use a service defined by the Name “services/getWeather” and this service is provided by two Producer P_1 and P_2 as seen in Figure 3.1-1. In order to establish the session, the Consumer C has to create an Interest which will request a session from one of both Producers.

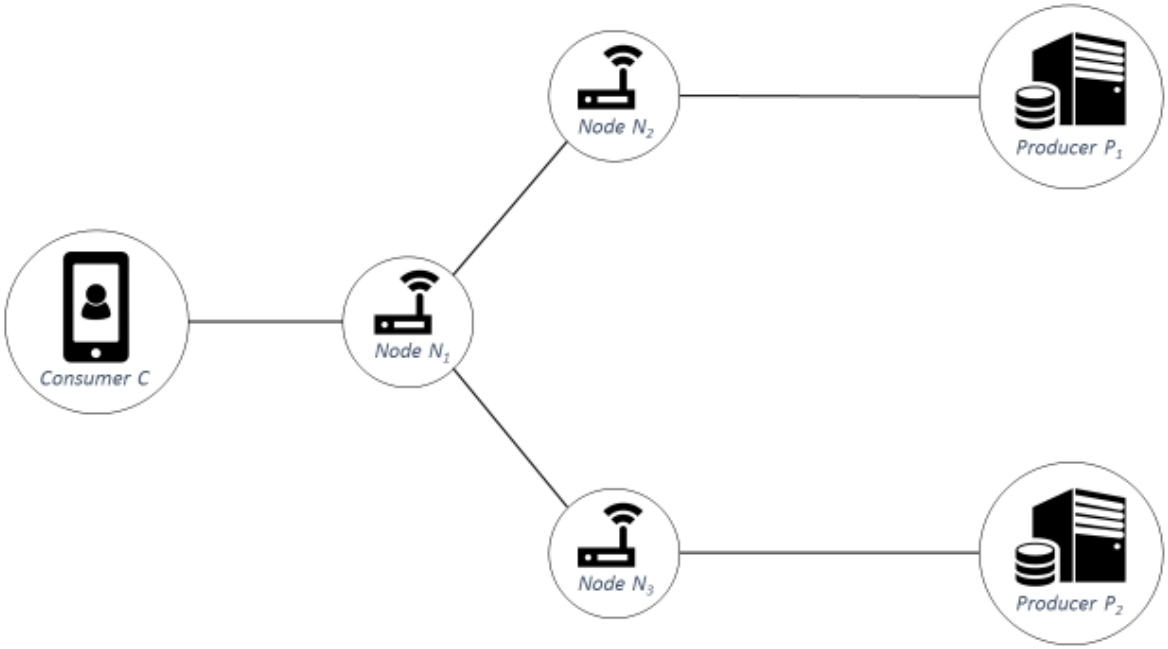

Figure 3.1-1 The topology of the example containing one Consumer and two Producers, which provide the service defined by the Name “services/getWeather”.

In order to create this Interest, the Consumer C has to generate a Name which consists of three parts. The first part is the Name of the service that is being requested, which would be “services/getWeather” in our example. Secondly, an identifier denoting that this is a session request

is required in the form of “session/request”. Finally, the third part is a unique identifier for the requested service generated by the Consumer C, which in our example shall be the number “12”.

The result by putting all three parts together is the complete Name of the Interest “services/getWeather/session/request/12”. The Interest with the complete Name is finally sent to the Node N_1 due to the existing route in the FIB. The PIT is updated to reflect that the Consumer C sent the Interest as seen in Figure 3.1-3.

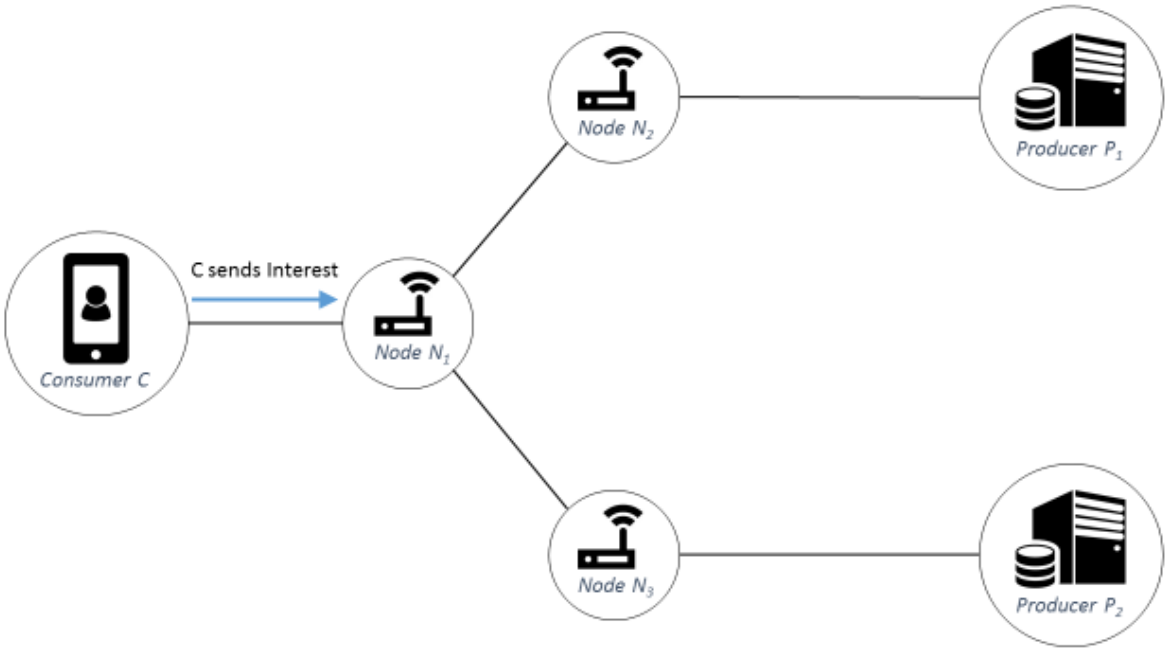

Figure 3.1-2 Consumer C sends Interest to Node N_1.

Consumer C

FIB	
Name	Node
services/getWeather	N_1

PIT	
Name	Node
services/getWeather/session/request/12	C

Figure 3.1-3 FIB and PIT entries of Consumer C after sent the Interest. Note that in the PIT the Interest was sent by the Consumer C, while the requesting Node of the Interest in the PIT is the Consumer C itself.

When the Node N_1 receives the Interest, it checks its FIB and finds two Nodes, to which it can forward the Interest as shown in Figure 3.1-4. In this example, we assume that N_1 uses the best-route strategy and hence sends it to the Node with the lowest routing cost. By assuming that this would be Node N_3, the Node N_1 therefore decides to forward the Interest to the Node N_3 as seen in Figure 3.1-5. (Analog, if the Node with the lowest routing cost was N_2, the Node N_1 would have forwarded the Interest to the Node N_2.)

FIB	
Name	Node
services/getWeather	N_2, N_3

Figure 3.1-4 FIB entry of Node N_1.

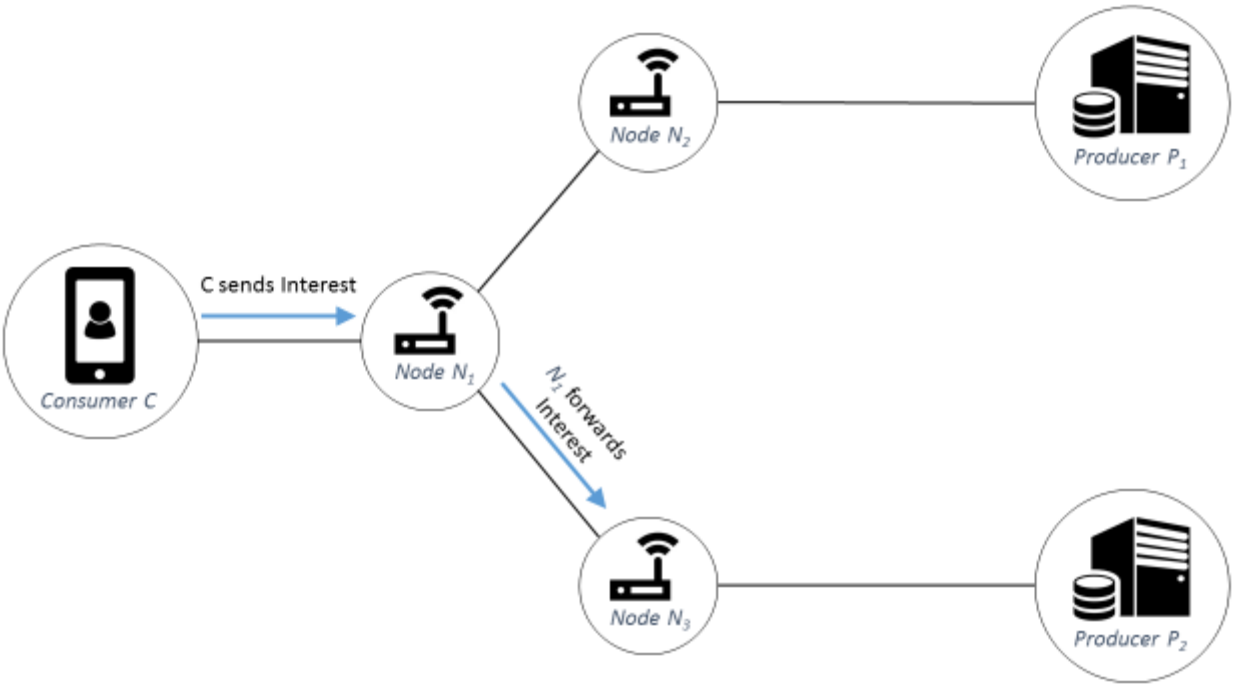

Figure 3.1-5 After the Consumer C sent the Interest to Node N_1, it is forwarded to Node N_3.

As Node N_3 is directly connected to the Producer P_2, which provides the requested service, the Interest is forwarded by N_3 towards P_2 as defined in its FIB. As a Session Identifier requires a unique identifier of the Consumer C as well as Producer P_2, P_2 generates its own unique identifier, which it stores into the Data of the response to the interest. This Data is then sent back to Node N_3 due to the PIT entry of the Producer P_2 as seen in Figure 3.1-6. The unique identifiers are randomly generated numbers of a specific length and the Session Identifier is the concatanation of two unique identifiers.

Producer P_2

FIB	
Name	Node
services/getWeather	P_2
services/getWeather/session/1298	P_2

PIT	
Name	Node
services/getWeather/session/request/12	N_3

Figure 3.1-6 FIB and PIT entries of Producer P_2. The new route has already been added to the FIB, before removing the PIT entry.

Before removing the PIT entry, the FIB is updated by adding a new route which allows Consumer C to connect to Producer P_2 by combining both unique identifier. The identifier of C is found in the Name of the returned Data, which is identical to the Name of the Interest stored in the PIT. The Identifier of the Producer P_2 is stored in the Data of the response. The new route can finally be added by using both Identifiers combined. Using this procedure the FIB of every Node traversed by the Data response adds the new route.

For example, when Node N_1 gets the Data from N_3, it adds the new route by combining the unique identifier of the Consumer C (from the Name) and the unique identifier of the Producer P_2 (from the Content) as seen in Figure 3.1-7.

Node N_1

FIB	
Name	Node
services/getWeather	N_2, N_3
services/getWeather/session/1298	N_3

PIT	
Name	Node
services/getWeather/session/request/12	C

Figure 3.1-7 FIB and PIT of Node N_1 when the Data is returned.

After the Consumer C has received the response to its interest, each node in the topology has stored the new route in the FIB which will be used by the session simillarly to the FIB of Node N_1 seen in Figure 3.1-7. The complete transition of the Interest and its returned Data is shown in Figure 3.1-8.

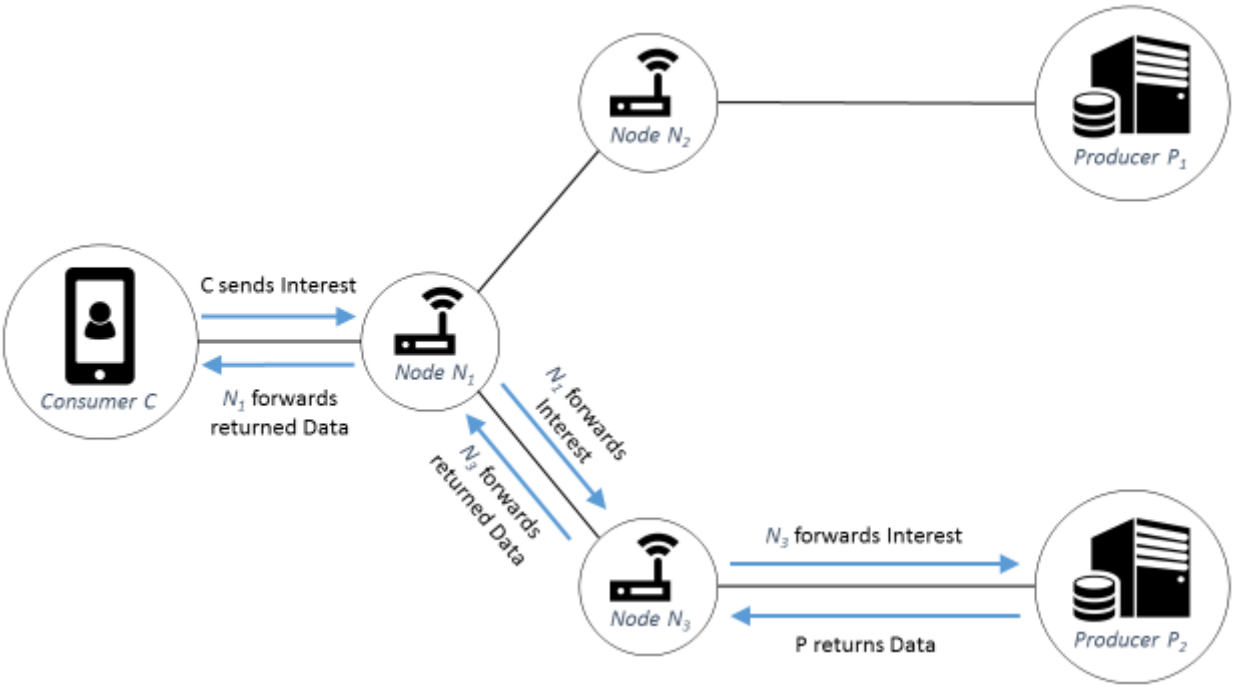

Figure 3.1-8 Complete transition of the Interest sent by Consumer C and the returned Data by P_2.

When the Consumer sends an Interest using the session, its Name needs several pieces of information: the Name has to contain the service, an identifier specifying that a session is used and finally the combined identifier. Hence, the name will be equivalent to the information stored in the FIB, which is “services/getWeather/session/1298” in the example above. This Name allows the Nodes to forward the incoming Interest towards the Producer based on the FIB.

3.2 Implementation

3.2.1 Implementation of the Design

The Session Support concept is written in C++ on a 64-bit machine running Linux Ubuntu 14.04 LTS. The IDE used was Eclipse 3.8.1, although there were no advantages to using an IDE except for code highlighting. Unfortunately, the current release of ndnSIM cannot be debugged in an IDE due to its complexity.

The same topology as the example given in Chapter 3.1 is used to develop the Session Support concept in ndnSIM. Therefore, the topology is set up by writing a simulation containing one Consumer, three Nodes and two Producers, which are connected as shown in Figure 2.1-2.

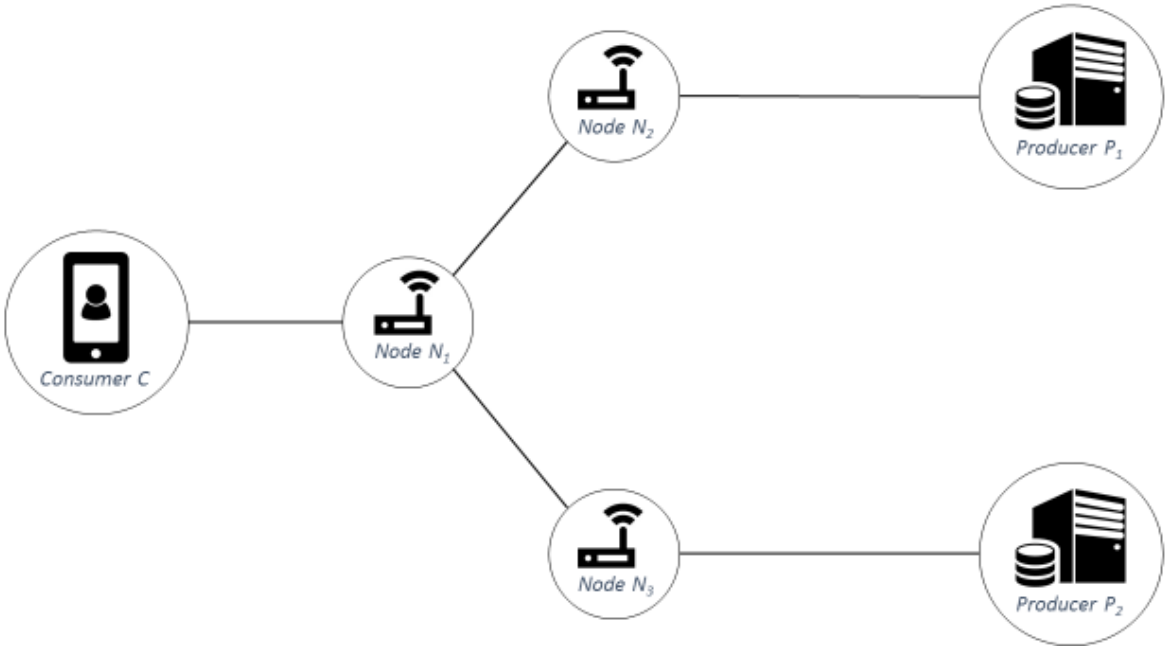

Figure 3.2-1 Identical topology as the example given in the previous chapter. It contains one Consumer, three Nodes and two Producers.

In ndnSIM every Consumer and Producer is also considered a Node. Hence, the amount of nodes defined is six as seen on line 63 of Figure 3.2-2. As this topology contains only a few Nodes, the connections were defined manually on the lines 67 to 71 of Figure 3.2-2.

```
61  // Creating nodes
62  NodeContainer nodes;
63  nodes.Create(6);
64
65  // Connecting nodes using two links
66  PointToPointHelper p2p;
67  p2p.Install(nodes.Get(0), nodes.Get(1));
68  p2p.Install(nodes.Get(1), nodes.Get(2));
69  p2p.Install(nodes.Get(1), nodes.Get(3));
70  p2p.Install(nodes.Get(2), nodes.Get(4));
71  p2p.Install(nodes.Get(3), nodes.Get(5));
```

Figure 3.2-2 Setting up the topology in ndnSIM.

Due to the connections defined between two Nodes, it is important to note that Node 0 represents the Consumer, Nodes 1 to 3 are the Nodes N_1 to N_3 while Nodes 4 and 5 embody the Producers P_1 and P_2. The next step is shown in Figure 3.2-3, in which the NDN stack is installed on all nodes and the forwarding strategy is chosen, which will be the *Random Strategy* in this simulation. This *Random Strategy* ensures that different Producers receive the Interest and hence it reduces the load on each one of them.

```
73  // Install NDN stack on all nodes
74  ndn::StackHelper ndnHelper;
75  ndnHelper.SetDefaultRoutes(true);
76  ndnHelper.InstallAll();
77
78  // Choosing forwarding strategy
79  ndn::StrategyChoiceHelper::InstallAll("/services/getWeather",
            "/localhost/nfd/strategy/best-route");
```

Figure 3.2-3 Installing NDN Stack on all Nodes and choosing the forwarding strategy.

For the Consumer, we need to define which implementation should be used as they can contain different attributes and functionalities. In this simulation we use the standard implementation "ns3::ndn::ConsumerCbr" as seen on line 84 of Figure 3.2-4, which contains only a few attributes and functionalities. The attribute "Frequency" sets the amount of Interests sent in a Second from the Consumer. Line 89 from Figure 3.2-4 defines which Name should be used for the Interest that the Consumer will send out at the Frequency defined above. Finally, all these parameters are installed onto Node 0, which represents the Consumer in the topology defined above.

```
83  // Consumer
84  ndn::AppHelper consumerHelper("ns3::ndn::ConsumerCbr");
85  // Consumer sends 1 interest per second
86  consumerHelper.SetAttribute("Frequency", StringValue("1"));
87  // Consumer will request /services/getWeather/session/request/wxyz
88  // where wxyz is a random generated Integer between 1111 and 9999
89  consumerHelper.SetPrefix("/services/getWeather/session/request/");
90  // Consumer is installed on node 0
91  consumerHelper.Install(nodes.Get(0));
```

Figure 3.2-4 Setting up a Consumer in ndnSIM.

In the current release of ndnSIM, there is only one implementation available for Producers, which uses a *Prefix* of a Name. The Producer will then reply to all Interests that contain the defined *Prefix*. As there are two Producers in our topology providing the same Service, only one *Prefix* is defined on line 97 of Figure 3.2-5 and then installed on both Nodes 4 and 5 representing the Producers.

```
93  // Producer
94  ndn::AppHelper producerHelper("ns3::ndn::Producer");
95  // Producer will reply to all requests starting
96  // with/services/getWeather
97  producerHelper.SetPrefix("/services/getWeather");
98  // First Producer is installed on node 4
99  producerHelper.Install(nodes.Get(4));
100 // Second Producer is installed on node 5
101 producerHelper.Install(nodes.Get(5));
```

Figure 3.2-5 Setting up both Producers in ndnSIM.

Now that the topology is complete, it is possible to run the simulation, but it will only consist of a regular NDN simulation.

As described in the Design of Chapter 3.1, the Consumer will have to send an Interest containing three parts: the requested service, a keyword showing that it is a request for a session and finally a unique identifier for the Consumer. The requested service as well as the keyword have already been

defined by the Name during the setup of the Consumer in the topology as seen in Figure 3.2-4: the requested service is defined by the *Prefix* “services/getWeather” and the keyword chosen for indicating that this is a request for a Session is “request” as shown on line 213 of Figure 3.2-6. The unique identifier of the Consumer will be a random Integer generated by the method from Figure 3.2-7.

```
210   // Check if Interest is connecting to a Service
211   if(interestName.find("services") != std::string::npos) {
212     // Check if Interest is requesting a Session
213     if(interestName.find("request") != std::string::npos) {
214       std::cout << "\033[1;35m" << "\nConsumer sends Interest with ";
215       std::cout << "Session Request..." << std::endl;
216
217       // Generate a unique identifier
218       uint64_t id = this->uniqueId();
219       nameWithSequence->append(boost::lexical_cast<std::string>(id));
220       std::cout << "\tName: " << nameWithSequence->toUri();
221       std::cout << std::endl;
222       std::cout << "\033[0m";
223
224       seq = 0;
225     }
...     ...
240   }
```

Figure 3.2-6 Consumer requests a Session.

```
162   uint64_t
163   Consumer::uniqueId()
164   {
165     // Create a random number between 1111 and 9999
166     int randomInteger = rand() % 8888 + 1111;
167     return boost::lexical_cast<uint64_t>(randomInteger);
168   }
```

Figure 3.2-7 The Consumer generates a unique identifier for its session request.

On line 219 of Figure 3.2-6 the unique identifier of the Consumer is appended to the Interest Name. If it is assumed that the random Integer is “1234”, then the complete Interest Name would be “services/getWeather/session/request/1234”.

When the Consumer has sent this Interest, the Nodes forward it to one of the two Producer depending on the Best Route Strategy. Upon receiving the Interest, the Producer checks if the Interest is requesting a Session for a Service. In this case the Producer generates a four digit unique identifier analog to the code of the Consumer in Figure 3.2-7 and stores this number in the Content of its Data Packet as shown on line 143 of Figure 3.2-8. This Data Packet is then returned to the Consumer based on the PIT entries as usual in NDN.

```
129    // Check if Interest is connecting to a Service
130    if(dataName.toUri().find("services") != std::string::npos) {
131      // Check if Interest is requesting a Session
132      if(dataName.toUri().find("request") != std::string::npos) {
133        std::cout << "\033[1;36m" << "\nProducer " << GetNode()->GetId();
134        std::cout << " received Session Request and creates ";
135        std::cout << "Data Packet..." << std::endl;
136        std::cout << "\tName:\t" << data->getName().toUri();
137        std::cout << std::endl;
138
139        // Generate a unique identifier
140        std::string ct = std::to_string(this->uniqueId());
141
142        // Store the unique identifier in the content
143        data->setContent(reinterpret_cast<const uint8_t*>(ct.c_str()),
144                         ct.size());
145        const ndn::Block& block = data->getContent();
146
147        // Read the content in order to confirm the unique identifier
148        std::string storedContent =
149            reinterpret_cast<const char*>(block.value());
150        storedContent = storedContent.substr(0, block.value_size());
151
152        std::cout << "\tContent: " << storedContent;
153        std::cout << "\033[0m" << std::endl;
154      }
...    ...
175    }
```

Figure 3.2-8 Producer responds to an Interest requesting a Session.

Each Node that receives the Data Packet and forwards it to the Consumer needs furthermore to add the new route for the Session into its FIB as described in the Design of Chapter 3.1. Therefore, each Node checks the incoming Data Packets whether the Name shows that a Session of a Service was requested. For the new route two pieces of Information are crucial: On one hand, the Name indicating the use of a Session including the Session Identifier is required and on the other hand the route needs to know where to forward this Name.

First, for the Name of the Session, the Node gets the partial Name of the Data Packet until and with the keyword “session” as seen on lines 335 to 345 of Figure 3.2-9, which would be “services/getWeather/session/” due to the Interest sent by the Consumer before. The Node then gets the Identifier of the Consumer from the current Name as shown on line 348 of Figure 3.2-9, which is “1234” as stated in the assumption above. The Identifier of the Producer is stored in the Data, which will be read by the Node on lines 351 to 353 of Figure 3.2-9. It is assumed that the generated unique Identifier of the Producer is “9876”. On line 356 of Figure 3.2-9 the Node combines both Identifiers in order to get the Session Identifier “12349876”, which it finally appends to the partial Name on line 360 of Figure 3.2-9 resulting in “services/getWeather/session/12349876”.

```
325    Name name = data.getName();
326    if(name.toUri().find("services") != std::string::npos) {
327      if(name.toUri().find("request") != std::string::npos) {
328        std::cout << "\033[1;33m";
329        std::cout << "\nForwarder received Data Packet" << std::endl;
330
331        // Start generating the name that will be used by the session.
332        Name newName("");
333
334        // Get the position of the keyword "session"
335        Name tmpName("session");
336        int pos = 0;
337        int nameSize = boost::lexical_cast<int>(name.size());
338        for (pos = 0; pos < nameSize; pos++) {
339          if (name.at(pos) == tmpName.at(0))
340            break;
341        }
342
343        // Create new Name for Session, by getting Name until and
344        // with the position of  keyword "session".
345        newName.append(name.getSubName(0, ++pos));
346
347        // Get identifier of consumer from Data Packet Name
348        std::string consumerId = name.at(pos+1).toUri();
349
350        // Get identifier of producer from Data Packet Content
351        const ndn::Block& block = data.getContent();
352        std::string ct = reinterpret_cast<const char*>(block.value());
353        std::string producerId = ct.substr(0, block.value_size());
354
355        // Generate combined identifier
356        std::string sessionId = consumerId + producerId;
357        std::cout << "\tCombined Identifier: " << sessionId << std::endl;
358
359        // Complete new Name of Session by appending sessionId.
360        newName.append(sessionId);
...        ...
375      }
376    }
```

Figure 3.2-9 Each Node creates the Name of the new route for the Session.

Second, the route needs to know where to forward any Interests requesting the Name generated above. As the Node has just received a Data Packet from the Producer providing the requested Session, the Node can simply use the incoming Node of the Data Packet as the outgoing Node of the new route as shown on line 368 and 369 of Figure 3.2-10.

Finally, the Node can store the new route into the FIB. By inserting first the Name of the route for the Session on line 365 and 366 of Figure 3.2-10 and then adding the outgoing Node on line 371 of Figure 3.2-10.

```
364         // Store new Name in FIB, so that the Session can be used.
365         std::pair<std::shared_ptr<nfd::fib::Entry>, bool> newEntry =
366             m_fib.insert(newName);
367
368         shared_ptr<Face> incomingFace =
369             const_pointer_cast<Face>(inFace.shared_from_this());
370
371         newEntry.first->addNextHop(incomingFace, 0);
```

Figure 3.2-10 Node stores the new route in FIB for using a Session.

When the Consumer receives the returned Data Packet from the Producer, it stores the Name of the new route for the Session in its FIB like the Node above and additionally stores it in a global variable called “sessionName”. When a Consumer establishes a new Session, this variable “sessionName” has to be reset as seen on line 302 of Figure 3.2-11 in order to store only the newly established one. It then looks for the keyword “session” and copies the complete name until the keyword “session”, which would be “services/getWeather/session/”. Afterwards it extracts the Identifier of the Consumer “1234” from the Data Packet Name on line 318 of Figure 3.2-11 as well as the Identifier of the Producer “9876” from the Data Packet Content as seen on lines 321 to 323 of Figure 3.2-11. By combining both Identifiers, the Session Identifier “12349876” is generated on line 327 of Figure 3.2-11 and it is finally appended to the Name until the keyword “session” to result in “services/getWeather/session/12349876” on line 330 of Figure 3.2-11. The Name for using the Session is stored as a global variable in the Consumer and can be used henceforth.

When the Consumer decides to use the Session, it takes the Name stored in the global variable “services/getWeather/session/12349876” for its Interest. The next Node receiving this Interest can then forward it by using the information stored in the FIB until reaching the Producer connected to this Session. When the Producer finally receives this Interest it sends its response which follows the PIT entries that have been generated by the Interest until finally arriving again at the Consumer.

```
292    std::string name = data->getName().toUri();
293    if(name.find("services") != std::string::npos) {
294      if (name.find("request") != std::string::npos) {
295        sessionRequest++;
296        std::cout << "\033[1;35m";
297        std::cout << "\nConsumer received Data Packet...";
298        std::cout << std::endl;
299        std::cout << "\tName:\t" << data->getName().toUri();
300        std::cout << std::endl;
301        // Reset Name to be used by Session
302        sessionName.clear();
303
304        // Get position of keyword "session"
305        Name tmpName("session");
306        int pos = 0;
307        int nameSize = boost::lexical_cast<int>(data->getName().size());
308        for (pos = 0; pos < nameSize; pos++) {
309          if (data->getName().at(pos) == tmpName.at(0))
310            break;
311        }
312
313        // Create new Name for Session, by getting Name until and
314        // with the position of  keyword "session".
315        sessionName.append(data->getName().getSubName(0, ++pos));
316
317        // Get identifier of consumer from Data Packet Name
318        std::string consumerId = data->getName().at(pos+1).toUri();
319
320        // Get identifier of producer from Data Packet Content
321        const ndn::Block& block = data->getContent();
322        std::string ct = reinterpret_cast<const char*>(block.value());
323        std::string producerId = ct.substr(0, block.value_size());
324        std::cout << "\tContent: " << producerId << std::endl;
325
326        // Generate combined identifier
327        std::string sessionId = consumerId + producerId;
328
329        // Complete new Name of Session by appending sessionId.
330        sessionName.append(sessionId);
331
332        std::cout << "\tNext Request:" << "\t" << sessionName.toUri();
333        std::cout << "\033[0m" << std::endl;
334      }
...          ...
370    }
```

Figure 3.2-11 Consumer stores the Name to use a Session.

By running the simulation requesting and using a Session, the output displays messages in different colors. Messages produced by the Consumer are Magenta, those regarding the Producer are Cyan and finally those displayed by any regular Nodes (Consumer and Producer are also considered regular Nodes) are Yellow.

As shown in the screenshot of a terminal window running the simulation in Figure 3.2-12, the Consumer first displays that it intends to establish a Session by sending an Interest with a specific Name containing the unique identifier of the Consumer “6920”, which was generated randomly. This Interest is forwarded to the Producer on Node 4, which adds its own unique Identifier “6318” to the content. When it returns this Data Packet every Node adds “NewName” to its FIB containing the combined Identifier “69206318”. Finally the Consumer receives the Data Packet and stores the Name for the Session in a global variable for its next requests. Finally, the Consumer uses the Session twice and the Producer on Node 4 replies as expected. This means that the Session was established and used successfully.

Compared to the standard NDN implementation, the following elements were added or changed for the implementation of the Session Support Concept:

- The Consumer, Producer and Forwarding Strategy need to check for keywords in the Interest Name. This additional feature allows the Consumer and Producer to generate a unique identifier when a Session is requested. For the Forwarding Strategy checking for keywords was implemented in order to add a new entry into the FIB.
- The Forwarding Strategy was altered in order to extract content from a data packet and store it in the FIB. This allows the Session to be forwarded to a specific Producer.
- Finally, the Consumer saves a Name of an Interest, which can then be used for the Session.

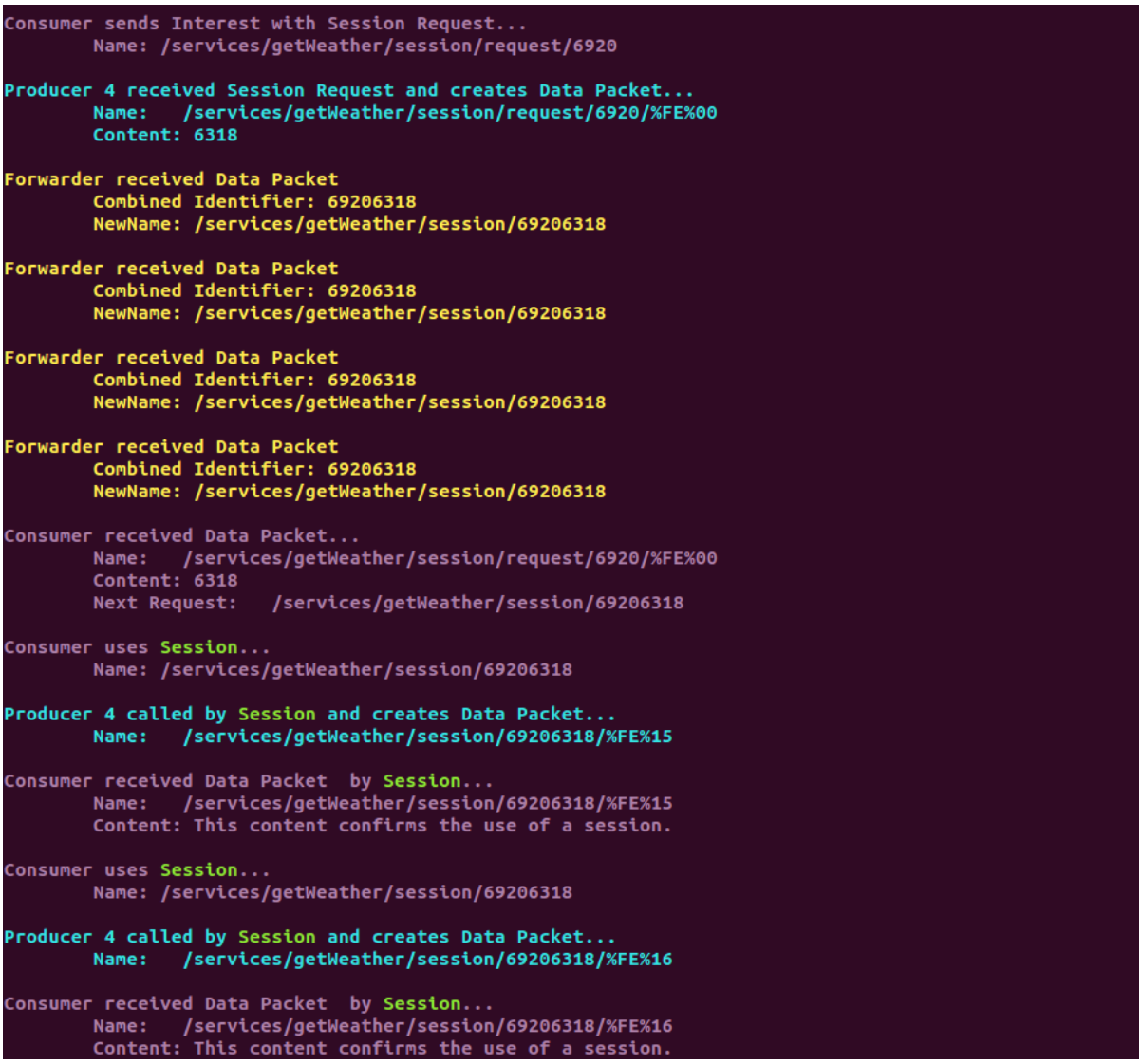

Figure 3.2-12 Screenshot of a terminal window displaying that a session is successfully established with Producer 4 and is used by the Consumer.

3.2.2 Implementation containing multiple Consumers

Before running any evaluations a second Simulation scenario was implemented in ndnSIM that contained several Consumers: The topology contained fifty Nodes including ten Consumers and eight Producers that were all connected randomly as shown in Figure 3.2-13.

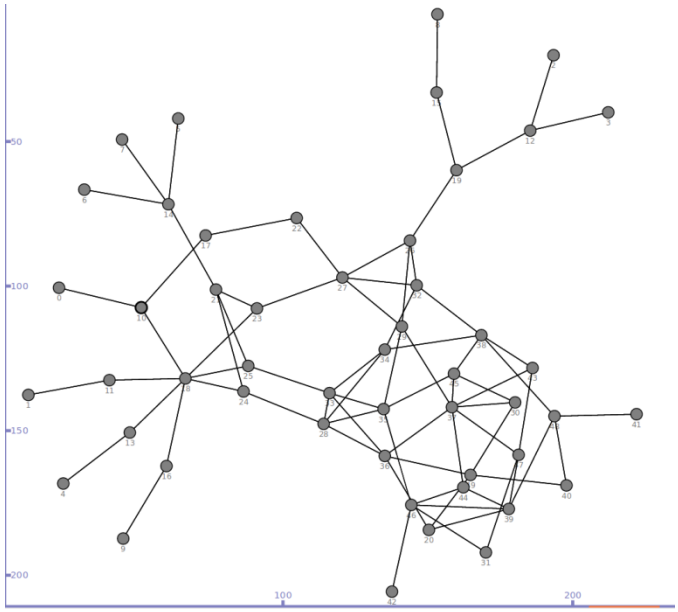

Figure 3.2-13 Topology with multiple Consumers.

In the Simulation, there were three Services defined by distinct Names: the Name of the first Service S_1 was "/services/getWeather", the second Service S_2 was defined by "/services/getNews" and finally the third Service S_3 could be requested with the Name "/services/getMedia".

Of the ten Consumers three requested S_1, another three S_2 and the last four tried to establish a Session with the Service S_3. Analogically, one Producer provided the Service S_1, two others delivered the Service S_2 and the last five Producers offered the Service S_3.

The Consumers sent out one Interest each every second. The first Interest sent by a Consumer requested a Session for one of the Services. After the Session was established the Consumer used the Session ten times and finally sent a request to tear down the session with its tenth Interest. This process was then repeated several times until the duration of the Simulation was over. The duration was set arbitrarily to 60 seconds.

When this new Simulation was run for the first time, there was an issue while using the Session: Some Consumers used a different Service than the one they had established a Session for. For example: A Consumer requested a Service from Producer 1, which correctly replies to set up the Session. Even though the Consumer successfully established the Session, it still used a Session provided by a different Producer, e.g. Producer 2. This was due to the fact that ndnSIM runs the same Instance of Consumer on every Node. Therefore, the Name stored in the global variable

"sessionName" was overwritten by every new Session that was established. By using an array of Names, in which every Consumer could store its own Name for the Session, the issue was removed. After the subsequent code was adapted to this issue, the Simulation ran correctly for 60 seconds.

The following example is an extract of this Simulation. The topology used is identical to the one described previously. The Consumer on Node 0, which is highlighted by a circle in Magenta in Figure 3.2-14, will request to establish a Session for the Service S_1 "/services/getWeather". Node 20 is the only Producer providing this Service and it should therefore reply. It is encircled in Cyan as seen in Figure 3.2-14. By checking the FIB and PIT entries of the Nodes reached by the Interest while being forwarded, it is possible to identify which Producer responds to this Interest and ensure that the returned data reaches again the Consumer on Node 0.

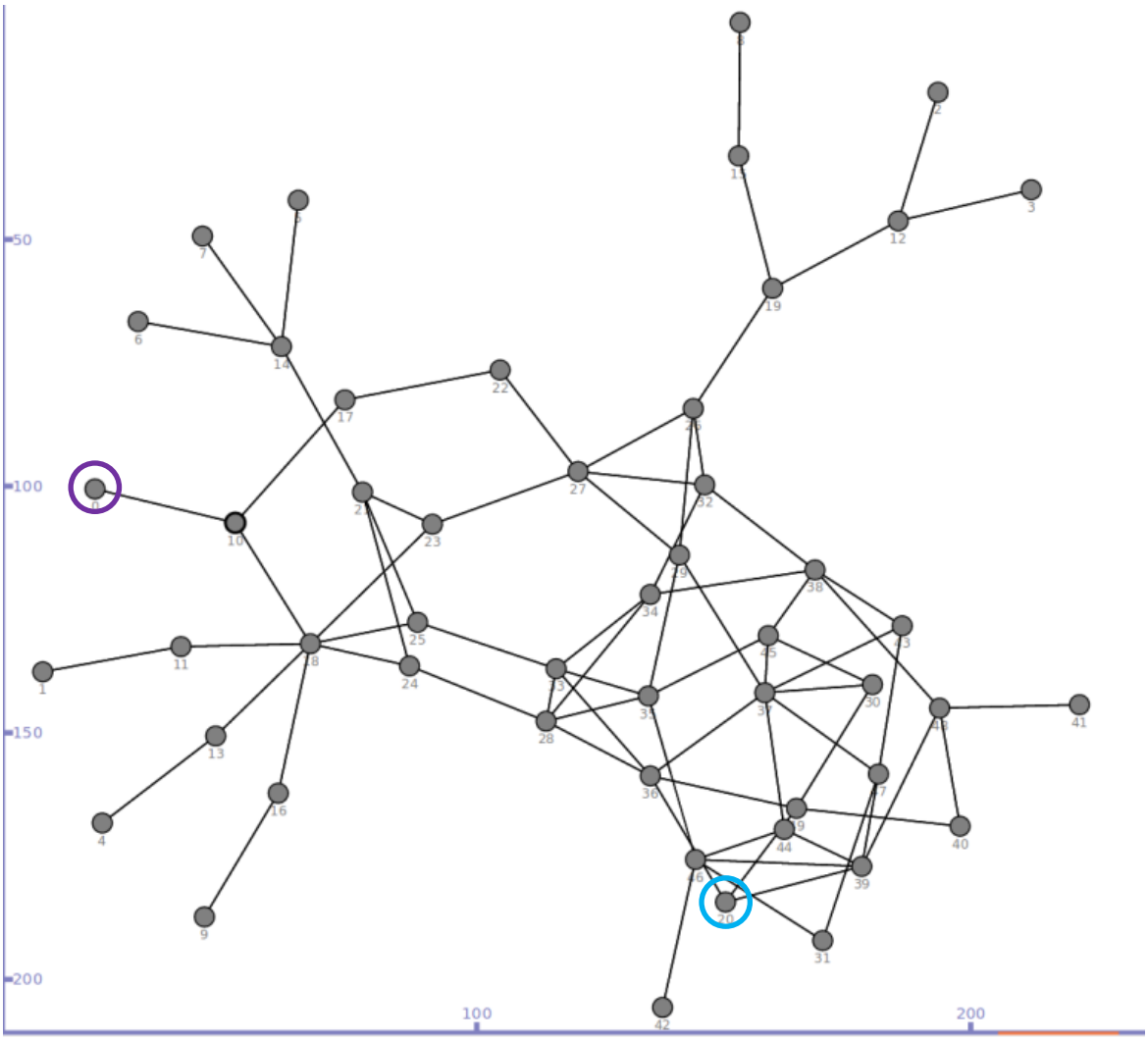

Figure 3.2-14 The topology of the example with the Consumer and Producer.

At the beginning of the Simulation, before anything happened, all Nodes have an empty PIT and almost identical entries in the FIB: Only the nodes used for the forwarding of an Interest are different.

When the Consumer on Node 0 sends an Interest with the Name "/services/getWeather/session/request/8718", the Interest is forwarded to Node 20 as shown by the green arrows in Figure 3.2-15 due to the FIB.

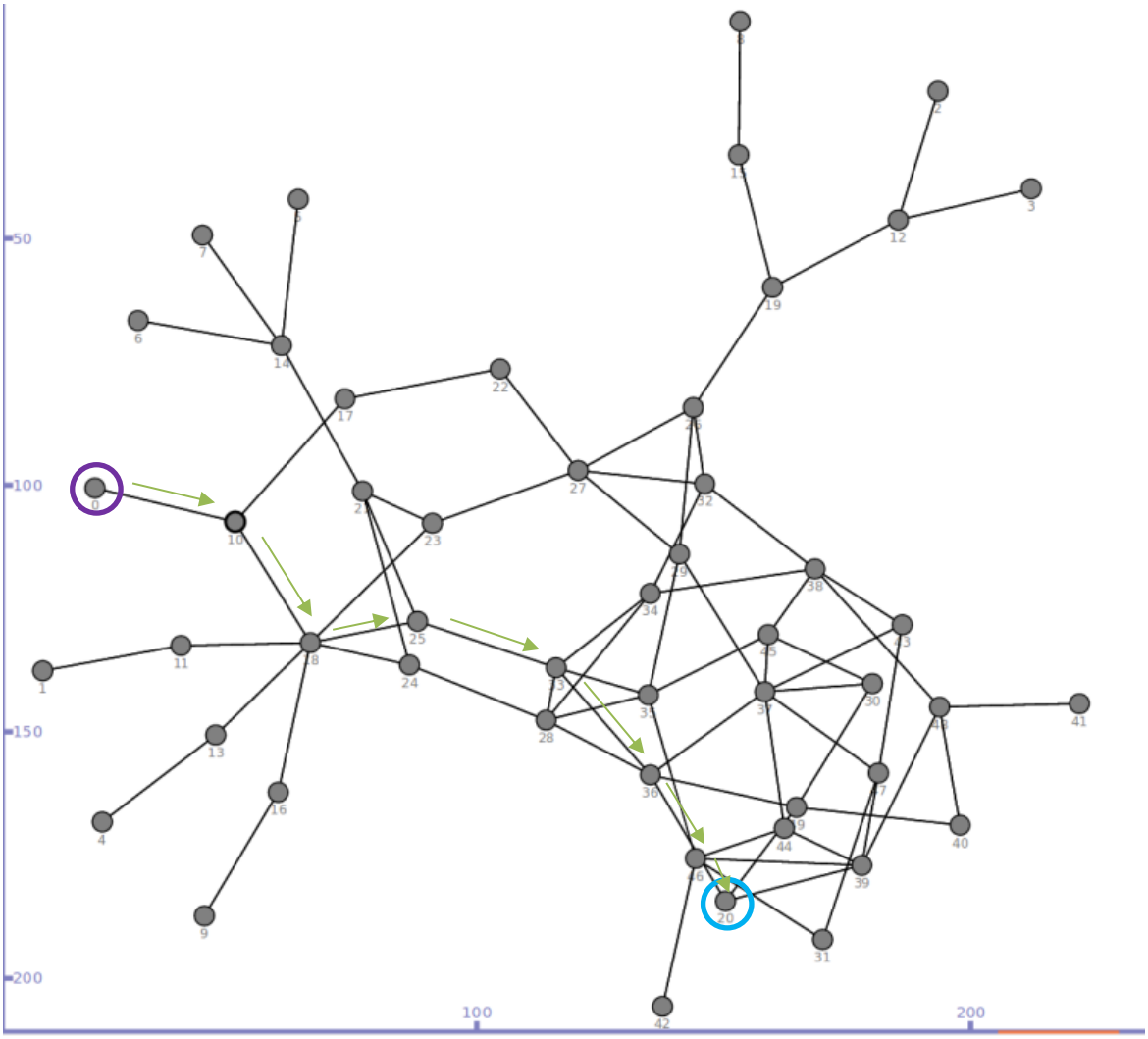

Figure 3.2-15 Forwarding of the Interest from the Consumer on Node 0 to the Producer on Node 20.

All Nodes have to update their PIT when forwarding the Interest. In Figure 3.2-16 the FIB and PIT entries of Node 18 are shown. The PIT entry written in red is only added when forwarding the Interest. For all other Nodes, which forward the Interest, the entries are identical with the following exception: The list of "Node" in the tables (FIB and PIT) are different. For example in the PIT of Node 10, the Node would be Node 0 instead of Node 10.

Node 18

FIB	
Name	Node
services/getMedia	23
services/getWeather	24, 25
services/getNews	23

PIT	
Name	Node
services/getWeather/session/request/8718/	10

Figure 3.2-16 FIB and PIT of Node 18: The PIT entry written in red is added when the Interest is forwarded.

Upon receiving the Interest, the Producer on Node 20 returns a Data Packet. This Data Packet is forwarded to the Consumer on Node 0 by traversing the PIT entries in the reverse order, which is shown by the red arrows in Figure 3.2-17.

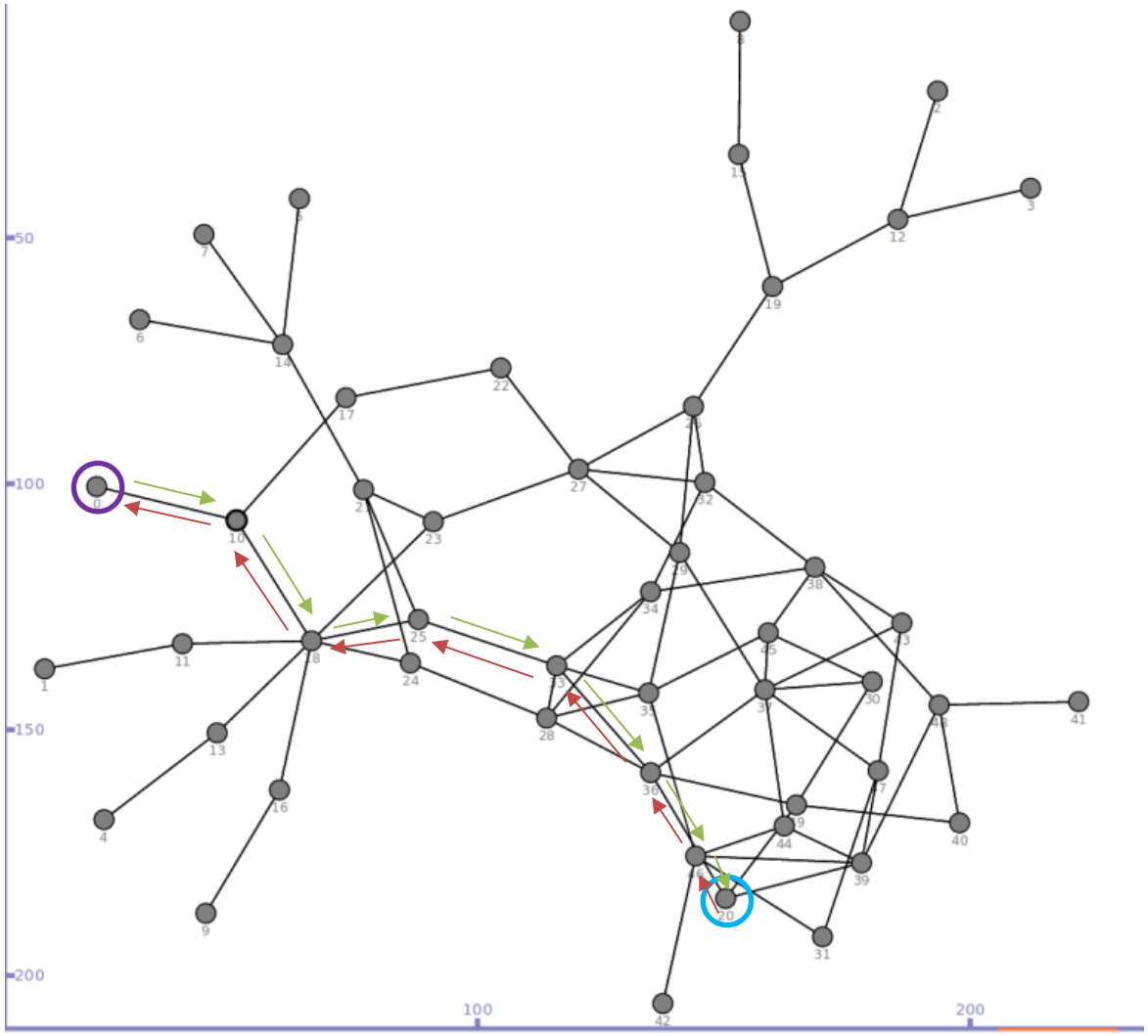

Figure 3.2-17 The Producer on Node 20 returns a Data Packet, which is forwarded to the Consumer 0 due to the PIT entries.

Every time a Node receives the Data Packet returned by the Producer, an additional FIB entry is added containing the unique Name for the Interest used by a Session. This can be seen exemplarily on Node 18, which creates the entry "/services/getWeather/session/87181162" in the FIB.

Node 18

FIB	
Name	Node
services/getMedia	23
services/getWeather	24, 25
services/getNews	23
services/getWeather/session/87181162	25

PIT	
Name	Node
services/getWeather/session/request/8718/	10

Figure 3.2-18 FIB and PIT of Node 18: The FIB entry written in red is the Name, which will be used by the Consumer for the Session.

As soon as Node 18 returns the Data Packet to Node 10, because of the entry in the PIT, this exact entry is removed. Hence, the resulting FIB only contains a single additional entry for the Session Support and the PIT is again empty as shown in Figure 3.2-19.

Node 18

FIB	
Name	Node
services/getMedia	23
services/getWeather	24, 25
services/getNews	23
services/getWeather/session/87181162	25

PIT	
Name	Node

Figure 3.2-19 FIB and PIT of Node 18: Resulting FIB and PIT after a Session was set up.

When the Consumer now uses the Session, it generates an Interest that contains the Name "/services/getWeather/session/87181162". Due to this Name and the previously added FIB entry, every Node knows exactly where to forward the interest. For example, Node 18 will forward such an Interest to Node 25, by using the information stored in its FIB.

Finally, the Consumer can send an Interest to tear down a Session. When a Node responds to this Interest, the FIB removes the previously added entry "/services/getWeather/session/87181162" to restore the FIB to its original state. This can be seen in Figure 3.2-20, which shows the FIB containing only its original entries and the empty PIT of Node 18 after the Session was torn down.

Node 18

FIB	
Name	Node
services/getMedia	23
services/getWeather	24, 25
services/getNews	23

PIT	
Name	Node

Figure 3.2-20 FIB and PIT of Node 18: The FIB and PIT states are back to their original state before a Session was set up.

4 Evaluation and Results

4.1 Testing Setup

The evaluation was based on the implementation with multiple Consumers. The random topology was identical as seen in Figure 4.1-1 and contained fifty Nodes, but included four Consumers (instead of ten) and eight Producers.

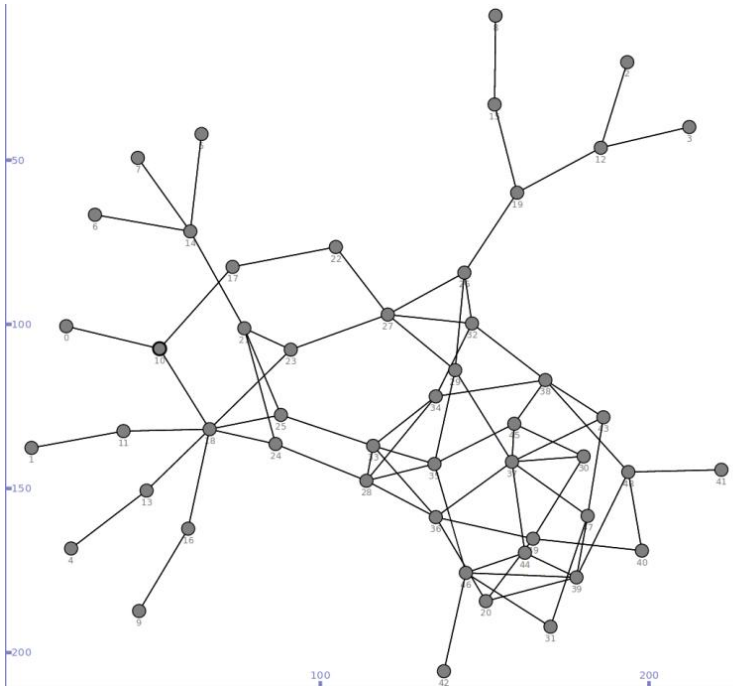

Figure 4.1-1 Topology for the evaluation.

For the evaluation four different implementations were run: One used the Session Support and the other three used the standard strategies available in ndnSIM. All simulations had in common that each Consumer sent out one Interest approximately every second and the duration of the simulations was set to 30 minutes. Those 30 minutes were chosen in order to contain at least 1000 unique interests, which would be used for a comparison of the simulations.

In the Simulation using the Session Support, the first Interest sent by a Consumer requested a Session for one of the three available Services. After the Session was established the Consumer used the Session two times and then sent a request to tear down the session with its third Interest. This process was repeated several times until the duration of the Simulation was over. The processing time for the Producers was a random uniformly distributed variable between 1.0 and 1.3 seconds.

In all the other Simulations, the Consumer sent Interests without requesting a Session. In those cases, the Producer would have to get the updated data from a different producer and therefore needs twice as much time compared to using a Session. Therefore, the processing time for the Producers was set to be a random uniform variable between 2.0 and 2.6 seconds.

4.2 Simulation Results

By running each of the four Simulations ten times, a dataset was created which can be reproduced. Based on this dataset the following two graphics were generated by using the statistics tool R. This evaluation compares the implementation of the Session Support to the three default Strategies in ndnSIM: *Random*, *Best Route* and *Multicast*. The default Strategies were chosen, because there are currently no implementations of a Session Support beside the one developed in this Bachelor Project.

The Figure 4.2-1 with boxplots displays the processing times for each Strategy. The wide horizontal bar in the boxes displays the median of the processing times, the box itself shows the lower and upper quartile of the processing time and the whiskers denote the minimum and maximal values of processing time excluding outliers. Finally, the wide vertical bar represents the outliers.

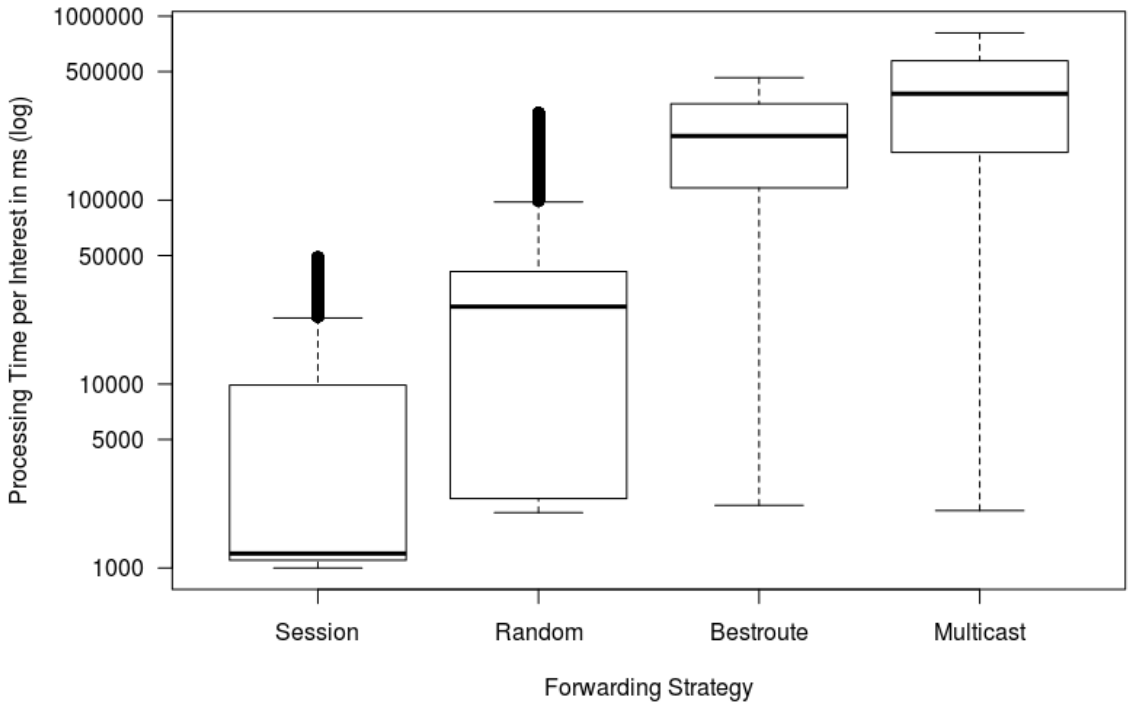

Figure 4.2-1 Processing Time per Interest for each Forwarding Strategy including the outliers.

This Figure 4.2-1 shows that when using a Session, the requests are handled almost immediately. On the other hand the other strategies contain a short delay before starting. The *Random strategy* is capable to process the interests much more efficiently then *Best Route* or *Multicast*, because the load is balanced randomly by this strategy. On the other hand, *Best Route* is capable to do a little bit better than *Multicast* because only one of the available Producers will receive the Interest. In Multicast all Producers receive all Interests, wherefore each Interest needs to be processed by every Producer. Subsequently, this results in a continuous increase of processing time for every Producer.

Therefore, the upper quartile and the maximum for the *Multicast* strategy are even higher compared to the *Best Route* strategy.

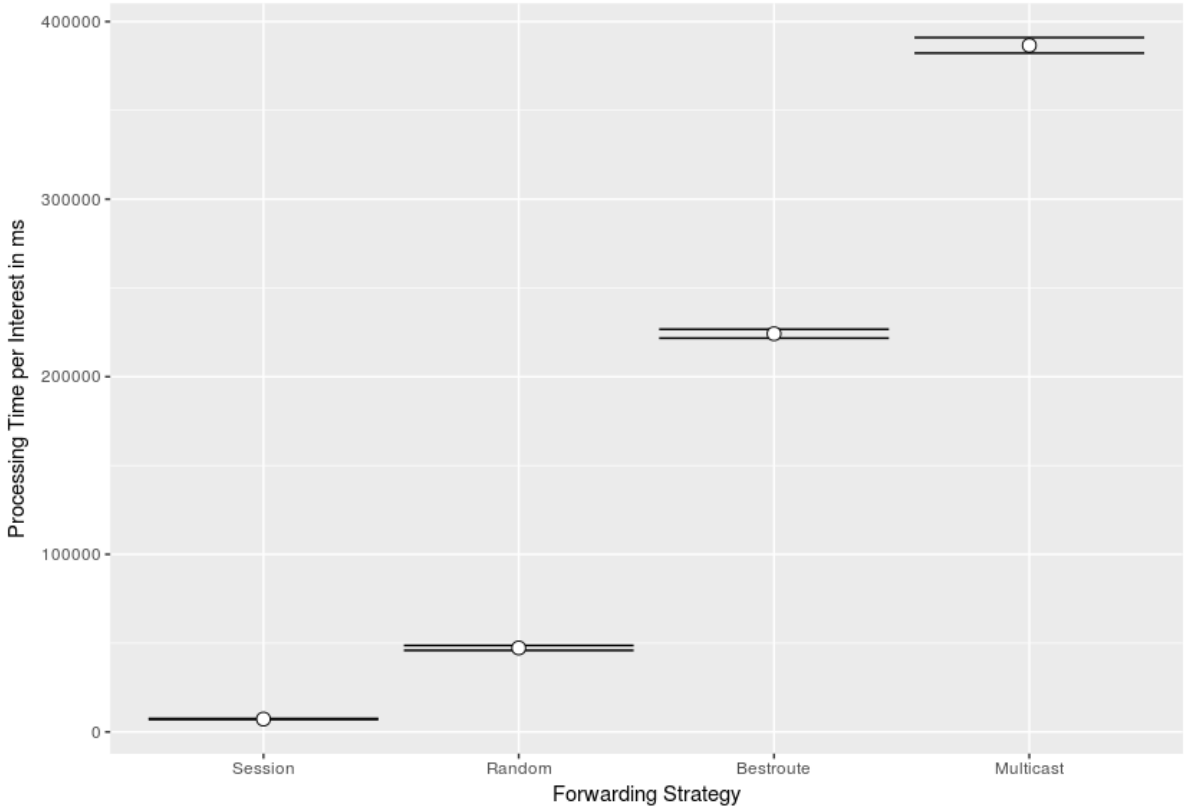

Figure 4.2-2 Arithmetic mean of processing Times and 95% Confidence Intervals for each Forwarding Strategy.

Figure 4.2-2 shows the arithmetic mean processing time and the 95% Confidence Interval. This Figure reflects clearly the advantage of the Session Support versus the other three Strategies. The implementation with the Session Support has demonstrated a mean processing time for the thousand requests of 7 Seconds (± 0,2 Seconds). The *Random Strategy* is the second best option in a Situation that requests a Session as it takes just a little bit more processing time. On the other hand, the Strategies *Best Route* and *Multicast* increase the processing time dramatically. This is due to the fact that these two strategies lead to a stagnation of interests, because they fail to do load balancing and hence don't have enough time to process them.

5 Conclusion

In this Bachelor Project a Session Support Concept was successfully implemented and evaluated in the simulator ndnSIM. The first implementation enabled one Consumer to set up and use a Session, but it failed to work with multiple Consumers. Therefore, the Consumer was enabled to store the correct Name to be used for the Session and to tear down the Session by sending an Interest containing a specific keyword. The Session Support was enabled without contradicting any of the key principles of NDN. Additionally, the advantages of Service-Centric Networking are still valid with the approach proposed in this Bachelor Project.

The evaluation has shown that a Session Support can lead to a significant reduction of processing times for Producers and hence reduce the amount of time that a Consumer waits for a response. The *Random* strategy would be the second best option, while the *Best Route* and *Multicast* Strategies need exponentially more time due to missing load balancing. Another advantage is the fact that a Session is available, allowing the Consumer to communicate with a specific Producer. This could also be the first part of an online identification.

As a Session creates a specific path from a Consumer to a Producer, Failure Resistance becomes very important. This could be developed in a future work focusing especially on the consequences of a failed node and a congested FIB. When a node fails, the path could be fixed either by searching for a new path to the Producer or alternatively by setting up a new Session with a different Producer. Regarding the congested FIB, the Session could be automatically torn down after a specific time, instead of using only explicit requests by a Consumer. Finally, an alternative concept without keywords could be developed since the current implementation relies heavily on keywords.

Literature

References

[1] *Content-Centric Networking - PARC, a Xerox company.* [Online] Available: http://www.parc.com/services/focus-area/content-centric-networking/. Accessed on: Aug. 29 2016.

[2] *NDN Frequently Asked Questions (FAQ) - Named Data Networking (NDN).* [Online] Available: https://named-data.net/project/faq/#How_does_NDN_differ_from_Content-Centric_Networking_CCN. Accessed on: Dec. 14 2016.

[3] Wikipedia, *Content centric networking - Wikipedia, the free encyclopedia.* [Online] Available: https://en.wikipedia.org/wiki/Content_centric_networking. Accessed on: Aug. 30 2016.

[4] *CCNx Overview | CCNx.* [Online] Available: http://blogs.parc.com/ccnx/what-is-ccn/. Accessed on: Sep. 08 2016.

[5] *Named Data Networking: Executive Summary - Named Data Networking (NDN).* [Online] Available: https://named-data.net/project/execsummary/. Accessed on: Jul. 21 2016.

[6] L. Zhang *et al.,* "ACM SIGCOMM Computer Communication Review: Named Data Networking," vol. 44, no. 3, pp. 66–73, 2014.

[7] V. Jacobson, "Introduction to Content Centric Networking," Bremen, Germany, Jun. 22 2009.

[8] *Interest Packet — NDN Packet Format Specification 0.2-alpha-3 documentation.* [Online] Available: http://named-data.net/doc/ndn-tlv/interest.html. Accessed on: Sep. 08 2016.

[9] *Named Data Networking: Motivation & Details - Named Data Networking (NDN).* [Online] Available: https://named-data.net/project/archoverview/. Accessed on: Jul. 21 2016.

[10] *new CCNx 1.0 binary release (20151104) | CCNx.* [Online] Available: http://blogs.parc.com/ccnx/2015/11/05/new-ccnx-1-0-binary-release-20151104/. Accessed on: Sep. 05 2016.

[11] *Libraries / NDN Platform - Named Data Networking (NDN).* [Online] Available: https://named-data.net/codebase/platform/#Version_050. Accessed on: Dec. 14 2016.

[12] T. Braun *et al.,* "Service-Centric Networking," [2010].

[13] *Reference Model for Service Oriented Architecture 1.0*, 2006.

[14] Wikipedia, *Session (computer science) - Wikipedia, the free encyclopedia.* [Online] Available: https://en.wikipedia.org/wiki/Session_(computer_science). Accessed on: Aug. 30 2016.

[15] Q. Wu *et al.,* "SOFIA: Toward Service-Oriented Information Centric Networking," *IEEE Network*, vol. May/June, pp. 12–18, 2014.

[16] *ndnSIM documentation - introduction.* [Online] Available: http://ndnsim.net/2.1/intro.html. Accessed on: Aug. 29 2016.

[17] *ndn-cxx: NDN C++ library with eXperimental eXtensions — ndn-cxx: NDN C++ library with eXperimental eXtensions 0.4.1-77-g667370f documentation.* [Online] Available: http://named-data.net/doc/ndn-cxx/current/. Accessed on: Aug. 30 2016.

[18] *Forwarding Strategies — ndnSIM documentation.* [Online] Available: http://ndnsim.net/2.1/fw.html. Accessed on: Sep. 07 2016.