UNIVERSITÄT BASEL

Computing the Distribution of Computations for Named Function Networking Using Name Based Routing

Masterthesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Computer Network Research Group
cn.cs.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Manolis Sifalakis

Christopher Scherb
christopher.scherb@unibas.ch
09-054-545

July 31, 2014

UNI
BASEL

Acknowledgments

On the shoulders of giants.
I would like to thank Prof. Dr. Christian Tschudin for giving me the opportunity for this work. Special thanks goes to Dr. Manolis Sifalakis for supervising this work, for his great support, for his ideas and his reviews. I would also like to thank Basil Kohler who developed the service layer system, which is directly connected with my work. I thank him for the great teamwork in the last half year. Furthermore, I want to thank everyone who supported me in this work especially my parents and my brother for proof reading this work.

Abstract

In Named Function Networking (NFN) λ-calculus[1] enables a programmatic way of interacting with an information-centric network (ICN) as a single computer.
To enable the ICN to understand and handle expressions written in the λ-calculus, which are encoded in names, NFN proposes the integration of an abstract machine that couples β-reduction with ICN forwarding primitives, and uses ICN as a memory substrate[2].
The goals of this project are, first to integrate a Call-By-Name Abstract Machine[3] into the existing CCN-lite implementation. Second, to couple its operation with a forwarding strategy and thereby combining resolution-distribution of λ-expressions across the network.
In the end a NFN capable ICN network will be able to evaluate λ-functions and optimize the location of the evaluation in the network.
Extensions for parallel computing and better load distribution inside the network are developed to provide a stable and fast execution process of computations.
For testing, native and Omnet experiments are conducted. The functionality of the NFN is proven in different test scenarios.

Table of Contents

1 Introduction

Current network technologies were designed to interconnect computers and computational resources in general. Alternative approaches are more focused on the data itself. In this work a data-centric approach will be extended to enable arbitrary transparent computations in a network.

1.1 Todays Computer Networks

Today computer networks play a fundamental economical, societal and developmental role for individuals, organizations and companies around the world. The number of computer users grew immensely in the last decades and therefore also the number of internet users. Many applications access the internet to communicate with other applications or to provide additional information and services to users. More and more services are provided via the internet and the number of people using these is growing. The internet driven growth has also promoted the integration of existing technologies and services. For instance Internet Service Provider (ISP) provide phone communication via the internet. Smartphones load maps for navigation applications on demand. Online storage systems like Dropbox, Google Drive or Microsoft Skydrive allow users to share their files by giving others a link. These systems are also used to increase the storage capacity of mobile devices by uploading files to the storage system. The computation power of smartphones and other devices can be extended by Cloud services such as Amazon Web Service. Computation intensive programs can be accelerated by moving the computation to such a Cloud Computing service. Media distribution is another important stake holder of the internet. Online video platforms like Youtube and Netflix are generating an extremly high network load by distributing high quality video files. All these examples suggest that the vast majority of internet usage consists of data being distributed from a source to a number of users[4].
The content is served particularly for each requester and not broadcast as it is traditionally done by TV and radio program distribution companies.
However, in the 1980's, when the network protocols of the internet were designed, the focus was not on distributing content. At that time important tasks were to connect to a printer or to connect to another computer to perform tasks. Users around the world were connected

with a few big servers. The central goal was to communicate with other computers. For this purpose machine addresses were given to the computers. Protocols were designed to interconnect devices i.e. resource locations, so the network cannot detect if two users connect to a host to request the same data.
Today's usage is vastly different, but the protocols are still the same. Problems, created by these changes, were solved by creating and expanding infrastructure and not by updating the protocols for the new requirements. Name resolution systems were created to enable users to address computers by human readable names and not by machine names. Communicating with a name resolution system issues additional traffic[5]. For instance the Domain Name System (DNS) translates a Uniform Resource Locator (URL) to an Internet Protocol (IP) address.
The load on the network system is handled by increasing the capacities of the network and the backbone. Especially video and audio content distribution generates a lot of traffic. To reduce the load on the lines large companies - mainly such providing audio and video services - installed so called Content Delivery Networks (CDN)[6]. A CDN is a replication server system distributed to various locations. All servers of a CDN work together to serve the requests of the users. First new content is placed on the publisher's server. Then this content is mirrored on replication servers of the CDN. User requests are distributed over all replication servers. This way the load on one server is reduced. Since the replication servers are also spread worldwide, user requests are redirected to replication servers at different location in order to reduce the network load on one backbone. The content can be statically distributed or on demand i.e. it will be replicated if many users ask for it. Companies like Google own and manage their own CDN. Amazon CloudFront and Microsoft Azure CDN are commercial CDNs where users can rent CDN servers when they need the capacity.
Moreover today security is an important aspect in computer networks. With the increasing number of users and services the internet became attractive to malicious and criminal activities. To protect sensible data and passwords in the network the connection channels were secured. Furthermore the source location is verified to protect users from communicating with fake sources. But usually only the channel is secured and not the data it delivers.
In conclusion you can say that many problems exists in the structure of the current internet. These problems can be addressed by New Generation Networks (NGN)[7].

1.2 Information Centric Networking

There are different approaches how a NGN could look like. Information Centric Networking (ICN) is one of these approaches. It is designed to address the problems of today's network system and to adapt the network architecture to current network usage[8]. Instead of interconnecting devices, an ICN connects the user to the data. Consequently the user can focus on the data he wants to access instead of addressing the location where the data are stored. A data requester asks the network to deliver the content i.e. named data. The network resolves the name of the data and transfers them to the requester. Moreover forwarding and routing in ICNs is based on content names. However forwarding of content names may prove to be challenging since the number of data files normally is unbounded compared to the number of hosts in the network. Therefore forwarding tables would be very large. This problem already exists in today's internet, because the number of hosts is ginormous huge. To reduce the size of the forwarding tables a hierarchical structure for the machine addresses

was introduced. A larger number of files compared with the number of hosts aggravates this problem for ICNs. A solution in ICNs may be similar to the solution in today's networks. For example in CCNx[9] hierarchical routing and aggregation are used. For ICNs the hierarchy does not use the machine addresses but the components of the content names.
By handling named data directly the network can detect requests for the same content and combine them if the data requesters are locally adjacent. The data can be transferred once per combined request and then be distributed to all requesters. This way the network load can be reduced and network congestion can be avoided. Furthermore it is possible to cache data in the network to achieve faster reaction times for frequently requested or long lived content. In comparison this is not possible in network systems serving requests from the data source. At the same time the availability of the content is increased. For example the cache system could be integrated in special routers. This way the concept of CDNs is directly integrated into the network patterns and becomes directly available to all users.
Additional security could be implemented directly on the data level. This is important and necessary to enable an arbitrary data source i.e. to guarantee the authenticity of data served out of a cache system. Since the data is secured and not the channel the user can distinguish data he requested from faked data easily. This is not possible in connection based network systems like the current internet, where the channel is secured. The user has to trust the data delivered by the channel.
However there are still problems in ICNs. For instance real time content and transactional services may be difficult to handle in ICNs, although solutions have been proposed e.g. for Voice Communication in ICNs[10]. But this still is a research topic. Building hardware routers for high speed routing of content in ICNs is also a requirement for efficient usage and scalability[11].
Although until now current ICNs only deliver static data stored in the network, there is a possibility to extend this to serve the data transformed to the form the user wants it.

1.3 Named Function Networking

Often a user does not want to work with the data themselves, but wants to analyze, transform or change the data. Especially if the data set is very large, it will be inefficient to send the entire data over the network. In other cases certain forms of data may not be accessible or available, even though they are in some transformed forms (e.g. often with media). In all of these cases, it is more useful to compute and deliver the result by processing or transforming the data on the server, since the result may often be much smaller than the initial data set. This paradigm to move the computation to the data and not the data to the computation is often used in Big Data software. E.g. Apache Hadoop[12], a framework for salable distributed software, supports this paradigm.
Unfortunately current ICN implementations like Palo Alto Research Center's (PARC) CCNx[4] can only address static content. But resolving the name of the data is just a special case of resolving an expression on the data. By perceiving λ-expressions[13] as names, computations can be directly integrated in the name resolution semantics of current ICN implementations. Functions can be accessed like content in the network and applied to named data. So the ICN is extended to become a Named Function Network (NFN). The user will be enabled to interact with the network in a programmatic way. The goal of this work is to introduce dynamic or on the fly generated content and to implement an expression resolution system

for an ICN.
Using the ICN approach, accessing data in the network becomes similar to accessing data on a local file system. With the NFN extension the entire network takes characteristics of a local computer, i.e., an abstract machine, programmable by λ-expression. λ-expressions can express programs written in a functional programming language. This property makes it possible to compile a program written in an arbitrary functional programming language to a λ-expression and to run it in the network. Additional the network can interact with functions written in high-level programming languages to simplify programming.
For example a user wants to watch a high resolution video on a mobile device, which does not support the resolution of the video. Instead of downloading and converting the video the user can ask the network to deliver the video in a specific resolution. If the video is not available in that resolution, it will be computed on the fly and delivered to the user. Possibly the video was already requested by another mobile device user and so the converted video can be delivered from the cache of the network. Another example is if the original data cannot be served, because of their size or because of security reasons, in this case the computation needs to take place on the node, which stores the data.
Moreover today heavy computations and storage operations are often outsourced and computed in large cluster systems. Cloud computing services enable a user to rent computation power on demand. Programming cloud systems can be simplified using a NFN. Instead of distributing the computation manually, the network optimizes where the computation is executed and how it is distributed.
A typical scenario for cloud computing includes Map-Reduced applications, which are inspired by the map and reduce functions for parallelization of Lisp, the first functional programming language. The user specifies a map function and a reduce function and the data, on which the functions should be executed. It is up to the NFN to fetch the code for the execution, to access the data and apply the functions. Another use could be a sensor network. A controller wants to collect measured data from the sensor network. The measured data themselves are not important, but statistical data computed out of the measured data. The controller can specify how this statistical data are computed and the network will deliver the data. It is up to the network where the computation takes place.

In this work CCN-Lite - an implementation of the CCNx protocol - will be extended to a NFN. Furthermore an implementation of the Map-Reduce operator will be integrated.

1.4 Motivation

To implement a NFN details have to be worked out. This thesis works out problems and solutions for a NFN-implementation. In large computer networks there are lots of nodes, which can be used for computation. With existing frameworks it is often hard to develop programs, that run in such a network. The programmer has to decide how to distribute the program between the nodes. Frameworks like Map Reduce have been proposed to simplify this task by running the program, where the data are stored. However Map Reduce is only one specific case. Detailed forwarding and communication strategies are developed to enable nodes in a NFN to perform non-blocking and parallel operations for arbitrary programs. Therefore the NFN can perform Map Reduce operations and give users the possibility for simple data parallel operations.

2 Background

2.1 Information Centric Networks (ICN)

The basis for ICN systems has emerged within the TRIAD-Project[14]. The core idea of the TRIAD-Project was to create a next generation network architecture to address the problems of current networks.
ICNs adapt concepts of the TRIAD-Project. One of the key concepts of all ICNs is that the location of the data is transparent, so that the data can be delivered from every node holding the data. This is achieved by name-based forwarding. Furthermore ICNs decouple the content sender from the receiver applying a publish/subscribe semantic for the communication.

In this work we specifically look at one ICN, which is the CCNx-Protocol[9] incarnated by the CCN-lite[15] implementation.

2.1.1 CCNx

CCNx is a protocol for information centric networking developed by PARC[4][9]. The communication consists of two types of packets: interests and content packets (i.e. data packet), which will be discussed in Section 2.1.1.1.
A communication between two nodes in the network takes place through a face. A node maintains various data structures to support resolution and forwarding of names and enable communication, which will be described in Section 2.1.1.2
The communication in CCNx is receiver driven. A user propagates an interest message to a subset of its neighbored nodes in the network to ask for a content object. Every node receiving the interest checks if it can satisfy the interest by answering with the requested content object. Otherwise, it will forward the interest message to other nodes. The routing process will be explained in Section 2.1.1.3.

2.1.1.1 Message Types in CCNx

The format of CCNx messages is a binary XML format. All information are placed between XML tags.
An interest message mainly contains the name (or prefix) of the data. Additionally, an interest message contains selector fields. These fields are used to specify the desired content. For example a user can specify the publisher or the scope and can apply selectors. Figure 2.1 shows an example how an interest could look alike in not-binary XML.

```
<Interest>
  <Name>
    <Component>PARC</Component>
    <Component>TESTFILE</Component>
  </Name>
</Interest>
```

Figure 2.1: An interest for the content "$/PARC/TESTFILE$" written in not-binary XML

A content object carries the data that should be delivered and its name. Additional the content is signed by the publisher's private key to guarantee the authenticity. The signature is always part of the content object as well as additional information about how the signature can be verified (called $signed\_info$).
Content objects are split into segments. These segments are similar to Bittorrent chunks[16]. The content objects are transferred to the requester in these small segments. To get an entire content object the user must express interests in each segment.
Figure 2.2 shows the two packet types.

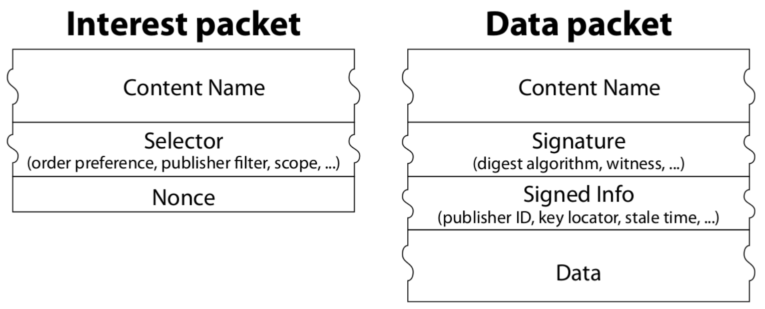

Interest packet
Content Name
Selector
(order preference, publisher filter, scope, ...)
Nonce
Data packet
Content Name
Signature
(digest algorithm, witness, ...)
Signed Info
(publisher ID, key locator, stale time, ...)
Data

Figure 2.2: CCNx packet types[4]

2.1.1.2 Data structures in CCNx

A CCNx node mainly maintains three important data structures: The Forwarding Information Base (FIB), the Pending Interest Table (PIT) and the Content Store (CS) or cache.
The FIB stores the information, which prefix can be reached, on which face. Therefore the FIB is similar to the FIB in current IP-Routers. It is possible that one prefix can be reached by multiple faces.

The PIT holds all interests forwarded by the node and the face, on which the interests arrived. This is necessary to route a content object on the reverse way of the corresponding interest message. By storing the interest messages in the PIT a node can detect and absorb duplicated interests. The PIT as well as the FIB entries can refer to more than one face, so that one content object can be distributed to various receivers. Entries in the PIT are deleted after a timeout.
The CS is the cache storage of a node. Each time a node forwards a content object this content object will be stored in the CS for a specific time. If other nodes ask for the same content object the interest can be satisfied directly. If the CS is full, an entry will be removed based on a replacement strategy. Some content objects can have a flag, not to be stored in cache. These content objects expire directly. For instance this is true for real time content.

2.1.1.3 Forwarding in CCNx

CCNx routes along the prefix names. The prefix with the longest match to the name in the interest will be chosen from the FIB.
If a node receives an interest message, it will first check its CS. If the content is already stored in the local cache, the node can answer directly. If not the node will check its PIT. If there already is an entry in the PIT for the interest the incoming face will be added to the existing entry. The interest will not be forwarded. Thus the interests are combined. If there is no entry in the PIT, the node will create a new entry in the PIT and will check the FIB how to forward the interest. To decide the face, to which the interest will be forwarded the entries in the FIB are compared with the name in the interest and the longest match will be used. If there is no match, the interest will be broadcast.
If a node receives a content object, it will check its PIT. If there is no entry in the PIT, the content object will be dropped. Therewith duplicated transmissions can be avoided. This is the case when the corresponding interest was already satisfied by receiving the content from another node or if there was no such interest. If there is an entry in the PIT, the content object will be forwarded to all faces on the list of the PIT entry and the entry is removed. This way combined interests are distributed to all data requesters and the network capacity is used more efficiently. Consequently the content object is routed on the reverse path of the interest. Additionally the node creates a new entry in its cache. Therefore further interests for the same content object can be satisfied directly. Figure 2.3 shows a simple schema of the CCNx forwarding engine.

2.1.2 CCN-lite

CCN-lite[15] is a lightweight implementation of the CCNx protocol developed by Tschudin and Sifalakis at the University of Basel. CCN-lite is developed for UNIX like system - mainly Linux - and is written in pure C and has a very lightweight core. Additional CCN-lite provides some command and control tools. CCN-lite can be deployed directly as kernel module for the Linux-Kernel. In this work CCN-lite is used as basis for the implementations.

2.2 The λ-Calculus

In a mathematical context a function is a relation between an input set and an output set, where each value from the input set is mapped to exactly one output. The λ-calculus

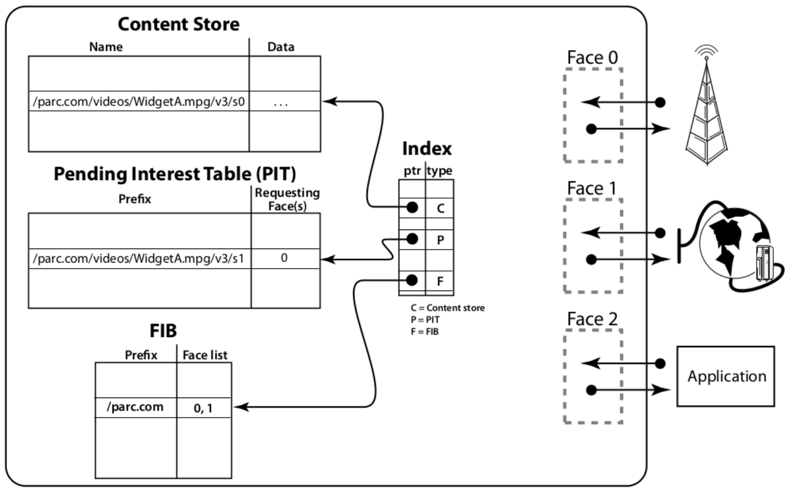


Figure 2.3: CCNx forwarding engine[4]

is a formal notation for functions based on mathematical logic. Functional programming languages are based on the λ-calculus or at least on the same idea, but normally they have additional constraints, e.g. for data types. It can be shown that the λ-calculus is Turing-complete[17].
In the following the untyped λ-calculus will be discussed but there is the possibility to extend the λ-calculus with different data types.

2.2.1 Expressions and Functions in the λ-Calculus

An expression in the λ-calculus (also simply called λ-expression) defined recursive as:

1. a variable x.
2. an abstraction $\lambda x.M$, where x is a variable and M is a valid λ-expression.
3. an application $M\ N$ were M and N are valid λ-expression.

and the corresponding grammar is:

$$M ::= x|M\ N|\lambda x.M, \tag{2.1}$$

So a mathematical function can be formulated in the λ-calculus. Normally functions in the λ-calculus are treated anonymously without an explicit name.
All operators and constants i.e numbers can be expressed as terms in the λ-calculus. Several ways to express constants and operators exist. One of these encodings is the Church's encoding[1], whereby prefix notation is used for operators. Prefix notation itself is bracket free, so that brackets are only used for special grouping. In the following there are a few examples how numbers and operators can be expressed in Church's encoding.
Natural numbers are defined as:

$$n = \lambda f.\lambda x.f^n\ x. \tag{2.2}$$

Hence zero becomes $0 = \lambda f.\lambda x.x$ and one becomes $1 = \lambda f.\lambda x.f\ x$.
One simple operator is the plus operator. Two natural numbers m and n can be added by

applying

$$m + n := \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x). \tag{2.3}$$

Analogically the multiplication of m and n is: $m \cdot n := \lambda m.\lambda n.\lambda f.m\ (n\ f)$ and the exponentiation m^n is: $m^n := \lambda m.\lambda n.n\ m$

A more complex operation is the $if\ else$ construct. For this operation it is necessary to define the constants $true$ and $false$: In Church's encoding true becomes $\lambda a.\lambda b.a$ and false becomes $\lambda a.\lambda b.b$. With this encoding $if\ else$ operations can be expressed as:

$$if(p)\ a\ else\ b := \lambda p.\lambda a.\lambda b.p\ a\ b \tag{2.4}$$

where p is the predicate (true or false), a will be executed if p is true and b will be executed if p is false.

Since it is very complex to express all numbers and operators in the λ-calculus the formal definition in Equation 2.1 can be extended with constants and operators[13]

$$M ::= c|x|M\ N|\lambda x.M, \tag{2.5}$$

where c is an operator or a constant. If c is an operator and not expressed in the λ-calculus, it can be directly executed by the underlying machine and does not need to be translated into a λ-expression. This might be a performance advantage, because the executing machine may support the operations in hardware.

This extension gives the possibility to simply formulate mathematical expressions, since the constants can be used as representation for numbers in the decimal system and there are higher order operators (e.g. $+, -, \cdot, /$): The function $f(y) = y + 3$ can now be expressed as $\lambda y. +\ y\ 3$ by using the prefix annotation. Without the extension this expression would be much more complicated:

$$\lambda y.(\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x))(y)(\lambda f.\lambda x.f\ (f\ (f\ x))), \tag{2.6}$$

where $\lambda f.\lambda x.f\ (f\ (f\ x)) = 3$ with the definition in Equation 2.3.

Beside the shorter and easier readable expression the extension permits named functions. A function can be declared as an operator. Thus complicated expressions can be simplified or expressions can be reused.

After defining expressions in the λ-calculus, the result can now be computed.

2.2.2 β-Reduction

In this section one will find a description how to compute a result of an expression formulated in the λ-calculus. There is only one rule of computation, which is called β-reduction. β-reduction consists of replacing the formal parameters of the λ-expression with the actual ones[13]. A β-reduction again results in a λ-expression, which may be a number in λ-calculus encoding (see Equation 2.2) or a higher order term (e.g. an addition with free parameters). Often different possibilities how to interpret a λ-expression exist and so the β-reduction can be executed in different orders[1]. Logically this leads to different intermediate results. But after several reduction steps the results will be the same. This property is called confluence. Confluence is defined by the $Church - Rosser\ Theorem$: "If a term M can be reduced (in

several steps) to terms N and P, then there exists a term Q, to which both N and P can be reduced (in several steps)"[13].
If no further reduction steps are possible for Q, Q will be called the normal form of M (Q is also the normal form of N and P).
A graphical scheme of this theorem is shown in Figure 2.4.

$$\begin{tikzcd}
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
N \arrow[dr, dotted, "*"'] & & P \arrow[dl, dotted, "*"] \\
 & Q &
\end{tikzcd}$$

Figure 2.4: Graphical example of the Church-Rosser Theorem[13]

The *Church − Rosser Theorem* points out a further property of the λ-calculus. Each λ-expression, which has a normal form, has exactly one normal form, since each β-reduction leads to the same result after a various number of reduction steps.
Furthermore there are λ-expressions, which do not reach a form where no further reductions are possible. Those expressions do not have a normal form. The simplest example for such an expression is: $(\lambda x.x\ x)(\lambda x.x\ x)$. This expression always reduces to itself.

An example shows how to reduce the addition of the numbers 1 and 2. The addition can be formulated by representing the numbers with Equation 2.2 and the addition with Equation 2.3:

$$\begin{aligned}
+\ 1\ 2 = {} & (\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x))(\lambda f.\lambda x.f\ x)(\lambda f.\lambda x.f\ (f\ x)) \\
\rightarrow \beta_1\ & (\lambda n.\lambda f.\lambda x.(\lambda f.\lambda x.f\ x)\ f\ (n\ f\ x))(\lambda f.\lambda x.f\ (f\ x)) \\
\rightarrow \beta_2\ & (\lambda f.\lambda x.(\lambda f.\lambda x.f\ x\ )\ f\ ((\lambda f.\lambda x.f\ (f\ x))\ f\ x) \\
\rightarrow \beta_3\ & \lambda f.\lambda x.(\lambda x.f\ x)((\lambda f.\lambda x.f\ (f\ x))\ f\ x) \\
\rightarrow \beta_4\ & \lambda f.\lambda x.(\lambda x.f\ x)(\lambda x.f\ (f\ x)\ x) \\
\rightarrow \beta_5\ & \lambda f.\lambda x.(\lambda x.f\ x)(f\ (f\ x)) \\
\rightarrow \beta_6\ & \lambda f.\lambda x.(f\ (f\ (f\ x)))\ =\ 3.
\end{aligned} \tag{2.7}$$

Equation 2.7 shows a step by step β-reduction of the addition $1\ +\ 2$ written in Church's encoding. One can easily see the confluence property e.g. the reduction steps β_4 and β_5 can be swapped so that:

$$
\begin{aligned}
& ... \\
\rightarrow \beta_3 \quad & \lambda f.\lambda x.(\lambda x.f\ x)((\lambda f.\lambda x.f\ (f\ x))\ f\ x) \\
\rightarrow \beta_4 \quad & \lambda f.\lambda x.(\lambda x.f\ x)(\lambda f.f\ (f\ x)\ f) \\
\rightarrow \beta_5 \quad & \lambda f.\lambda x.(\lambda x.f\ x)(f\ (f\ x)) \\
\rightarrow \beta_6 \quad & \lambda f.\lambda x.(f\ (f\ (f\ x)))\ =\ 3.
\end{aligned}
\tag{2.8}
$$

Instead of replacing the first input λf parameter in $(\lambda f.\lambda x.f\ (f\ x))\ f\ x$ by the actual parameter f it is also possible to replace the second parameter λx with the x.
With the extension shown with the grammar in Equation 2.5 an addition becomes simple and can be described by $\lambda m.\lambda n.\ +\ m\ n$. Consequently the addition of 1 and 2 is $(\lambda m.\lambda n.\ +\ m\ n)(1)(2)$. After two reduction steps it becomes a simple addition in prefix notation: $+1\ 2 = 3$.

2.2.3 Fix-Point Combinators

For arbitrary computation loops are a common concept, so it is necessary to formulate loops in the λ-calculus. Similar to functional programming languages there is no real loop. Instead loops can be expressed with the help of a fix-point combinator.
A fix-point combinator is a higher order function Y that computes the fix-point of other functions M. The other function M is one computation step of a loop iteration, whereas M does nothing to exit the iteration if the computation has finished. In Equation 2.9 the fix-point combinator Y is applied to a function M.

$$
\begin{aligned}
& YM \\
\rightarrow \beta \quad & M\ (Y\ M) \\
\rightarrow \beta \quad & M\ M\ (Y\ M) \\
& ... \\
\rightarrow \beta \quad & M\ M\ ...\ M\ (Y\ M).
\end{aligned}
\tag{2.9}
$$

Thus an anonymous recursive function is constructed, which can have the behavior of a `for` or a `while` loop in procedural programming languages. Various fix-point combinations for the λ-calculus exist. All combinators bring the same properties.
A well known fix-point combinator is the Y-combinator invented by Haskell B. Curry:

$$Y := \lambda f.(\ \lambda x.f\ (x\ x))(\ \lambda x.f\ (x\ x)). \tag{2.10}$$

For instance a function g is applied to the Y combinator:

$$\begin{aligned}
& (\lambda f.(\ \lambda x.f\ (x\ x))(\ \lambda x.f\ (x\ x)))g \\
\to \beta\ & (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) \\
\to \beta\ & g\ (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) \\
\to \beta\ & g\ g\ (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) \\
& ... \\
\to \beta\ & g\ g\ ...\ g\ (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x))
\end{aligned} \tag{2.11}$$

Equation 2.11 shows that the Y-Combinator has the same properties as a fix-point combinator described in Equation 2.9 since $(\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x))$ returns itself and adds a g at the beginning.
Alan Turing defined another fix-point combinator[13]:

$$Y := (\lambda x.\lambda y.y\ (x\ x\ y))(\lambda x.\lambda y.y\ (x\ x\ y)). \tag{2.12}$$

2.3 Abstract Machines

An abstract machine is a theoretical model of a computer. A well known abstract machine is the Turing Machine[18]. Today nearly all microprocessor systems are based on the "Von Neumann" architecture. Abstract machines can be used to execute programs written in a programming language, which do not fit the "Von Neumann" architecture well[19]. Functional Programming languages are an example for this. Additionally an abstract machine can be used to hide hardware details and make arbitrary hardware programmable the same way by running the abstract machine on top of the hardware. For instance *Sun* uses an abstract machine and created their programming language Java on top of it. *Sun*'s abstract machine is called Java Virtual Machine (JVM).
An abstract machine usually includes a stack and registers, similar to a microprocessor. It permits step by step execution of a program[19]. A program is a sequence of instructions taken from the instruction set of the abstract machine. The instruction set is also called intermediate language, because it is a layer between a high level programming language and the low level machine code. A common abstract machine has a program counter, which points to the instruction to be executed next. The program counter is advanced when the active instruction finished. The next position of the program counter is not necessarily the position of the next instruction, since jump instruction can move the program counter to an arbitrary position. If there is no further instruction to execute, the current program will terminate. This basic control mechanism is known as program loop[19].
Since functional programming languages are adopted from the λ-calculus described in Section 2.2 abstract machines for execution λ-expression exist. For instance the SECD machine[20] evaluates pure λ-expression.
In the following existing abstract machines for executing functional programming languages will be discussed.

2.3.1 Concepts of Abstract Machines for Functional Programming Languages

There are various abstract machines for evaluating functional programming languages, but all these machines have the same basic concepts[19]:

- A stack holds the intermediate results of a computation and pending function calls. It is also used to pass parameters to functions.
- An environment gives values to the variables.
- A code pointer points to the instruction yet to be executed.
- The tuple of a function body and an environment is called closure. A closure is used to represent a function as a value (a fundamental concept of functional programming languages).

2.3.2 Call by Value Abstract Machines

The first abstract machines for executing functional programming languages were Call by Value abstract machines[19], which evaluate function arguments exactly once before they call the function body. The Zinc Abstract Machine (ZAM)[21] as well as the SECD Abstract Machine[20] are Call by Value abstract machines that reduce pure λ-expression. In the following we will take a closer look at a ZAM. The computation configuration of a ZAM is a quad-tuple $< E,\ A,\ T,\ R >$, where E is the current environment, A is the argument stack, T is a term and R is the result stack. The environment E is a mapping from a bound variable of a term to the corresponding closure name. The argument stack A holds a sequence of closure names and is used to store intermediate results. A term T is a sequence of instructions and the result stack R is for interfacing with higher level languages and library functions during the evaluation of λ-expressions. The basic instruction set of a ZAM is[2]:

- ACCESS(n) - Lookup for the variable n in the environment E. Push the corresponding closure to argument stack A.
- CLOSURE(c) - Create a new closure using E and the term c and push it on the argument stack.
- GRAB(x) - Replace E with a new environment, which extends E with a binding between x and the closure found at the top of A
- TAILAPPLY - Pop a closure from the argument stack A and replace the current configuration's E and T with those found in the closure.

2.3.3 Call by Name Abstract Machines

A Call by Name abstract machine does not evaluate a parameter before a function is executed, but when the parameter is needed during the execution. An abstract machine with this property is also called lazy machine. This may be problematic, if a parameter is often used in a function, because it is evaluated every time it is used. This leads to a big overhead. Krivine's Abstract Machine[3] is a Call by Name abstract machine and like the ZAM and

the SECD Abstract Machine it executes pure λ-expression.
Furthermore Krivine's Abstract Machine can be implemented on top of the ZAM machine. To do this a resolve-by-name instruction RBN is introduced. There are three cases to handle depending on the structure of the λ-expression. The first case is a variable lookup (argument is a variable), the second case an abstraction (argument is a λ-term) and the third case is a function application (two arguments, first is a function, second is one or more variables/functions as argument for the first function). The Table 2.1 summarizes all three cases.

Table 2.1: Resolve by Name instruction with ZAM's machine[2]

Call	Case	Rewrite as
RBN(v)	Var	ACCESS(v); TAILAPPLY;
RBN(\x body)	Abstr	GRAB(x); RBN(body);
RBN(f g)	Apply	CLOSURE(RBN(f)); RBN(g);

2.3.4 Call by Need Abstract Machines

Similar to the Call by Name abstract machine a Call by Need abstract machine evaluates a function parameter when it is needed. The big difference between the Call by Name and the Call by Need abstract machine is that a Call by Need abstract machine maintains a heap[22]. The heap is used to store the results of the parameter evaluation. So a parameter is evaluated when it is needed first. If it is used again, the abstract machine will only do a lookup in the heap and will get the parameter value.

3 Related Work

3.1 Network Protocols

Mime (Multipurpose Internet Mail Extensions)[23] is a extension to the standard of the e-mail protocol. Mine enables nodes to define the type, the connection and the encoding for a data transmission by using additional header fields. This way it becomes possible to understand the encoding of transmitted text or to understand the transmitted data as video or picture and not as text. Mime encoding is also used to define the content type in the internet protocol HTTP[24]. In the following one can find an example for a Mine message:

```
MIME-Version: 1.0
Content-type: text/plain; charset=iso-8859-15
Content-Transfer-Encoding: 8bit.
```

Named Data Networking (NDN)[25] is another ICN protocol, which uses hierarchical routing similar as CCNx (see Section 2.1.1). The difference between CCNx and NDN is, that CCNx uses a binary XML packet format and NDN uses a TLV (Type-Length-Value) encoding. A TLV encoding is used to transmit a variable number of attributes in a message. The type determines the type of an attribute, the length defines the size of the attribute and the value contains the attribute itself. The functionality of NDN is more or less the same as the functionality of CCNx, the content distribution follows the publish/subscribe semantics and nodes use a FIB to forward interest messages which are stored in a PIT. A caching system for popular is also available.

3.2 Service and Network Level Programming

Service orientated Architecture (SoA) is a method to encapsulate existing services and network components and to combine the services to perform more complex operations. The concepts of SoA are used for workflow engines and business process management systems. For the communication between the services an arbitrary network protocol can be used.
A web service is a software application available over a network. To access a web service it is required that a Uniform Resource Identifier (URI) is provided. The URI contains information to identify the web service and a description of the interface, how to interact with

the service.
Web Services Business Process Execution Language (WS-BPEL) admits the specification of business processes (service composition) on the basis of the WSDL specifications. The process description is serialized using XML. Thereby various simple web services can be combined to complex business processes. WS-BPEL is not designed for interaction with humans, but only for service composition.
The Common Object Request Broker Architecture (CORBA)[26] is a middleware system built to define interfaces to services. Thus, it is possible to define access and composite services using CORBA. The core of CORBA are Object Request Broker. An ORB is a middleware used to call a function on another computer via a computer network. Such Remote Procedure Calls (RPC) are location transparent. To access a remote object CORBA defines the exchange format Interoperable Object Reference (IOR). CORBA contains a Naming Service to access remote objects by their names. The naming service return an IOR, which can be used to perform the RPC.
Similar as CORBA Java Enterprise Edition (Java EE) also provides the functionality for service composition and RPCs.
The Message Parsing Interface (MPI)[27] is an Application Programmable Interface (API) for High Performance Computing. MPI defines an interface for the communication between different processes. Thereby it does not matter if the processes run on a single machine or distributed over a cluster system. The MPI interprocess communication sends messages directly to one or multiple nodes in the network. Thereby the socket programming itself is hidden to the user, so that he can focus on the application instead of network programming. MPI is not designed to be used outside of HPC systems.
Apache Hadoop[28] is a framework for distributed and scalable software inside a cluster system. It is designed for data processing. Hadoop follows the Master/Slave approach where a master node manages a various number of storage nodes. Hadoop consists of a file system to store large data with high availability. The Hadoop File System (HDFS) is a distributed file system, which is designed to work with commodity hardware. This hardware is available for low costs, but is not reliable. Therefore, the file system has to be highly fault tolerant and must create replications of the files to reach a high reliability. To provide a reliable and performant system, it is very important to place the replication in the right location. The master node of the HDFS uses policies to determine where to place a replication. Thus, the location where data are stored is optimized. The transfer bandwidth of inter-rack is slower than intra-rack connections, but the probability that one rack fails is also higher than the probability that two dedicated racks fail. Therefore, a common probability is to store the one replication in the same rack and the second in another one. Hadoop implements the Map Reduce algorithm. Additionally, Hadoop has an API to execute functions on the node that stores the corresponding input data to reduce the network traffic inside the Hadoop Clusters.

3.3 Active Networking

Forwarding in a modern computer network consists of checking the packet header and applying predefined rules. Usually these rules consist of looking for matching entries in a FIB and applying them on those i.e. sending the packet to the next hop defined in the FIB.
Active Networking is an extension to classic forwarding. The goal is to make networks more

innovative, since the architecture of the current internet is extreme static. Routers and switches of an active network can perform customized computations on the packets[29].
Different approaches for active networking have been proposed. One of these approaches is Software Defined Networks (SDN). In a SDN a switch becomes programmable. The control plane of the switch will be outsourced into a controller. The controller is freely programmable. Based on the header information of a packet (as common switches do) and additional information (e.g. access authorizations) the controller can decide how to handle a packet. For instance Switchware[30] is a switch, in which the input and output ports are controlled by a programmable element. Openflow[31] is another implementation of a SDN. Additional to the programmable controller flows can be installed in an Openflow-switch. A flow is a forwarding rule defined by the controller. If a packet matches the flow, it can be directly forwarded without involving the controller to accelerate the forwarding procedure.
A second approach for active networking is that every message is a program composed of code fragments and the corresponding data. A node, that receives such a program, executes it. So the network itself becomes programmable. By injecting mobile code in the network it is possible to create new functionality. To achieve this it is necessary to have an encoding for network programs. Fraglets[32] are a unification of code and data. They are inspired by metabolistic pathway patterns found in biological cells. Fraglets are executed by applying predefined rules. Similar to real metabolistic processes a fraglet can react with another matching fraglet. So the matching rule combines two fraglets. Other execution rules are e.g. *send*, which sends the argument to another node or *split*, which breaks the fraglet into two smaller fraglets. For instance a fraglet can be used to send data from a node A to a node B and receiving and acknowledgement. Node A sends a fraglet to node B. When node B receives the fraglet it executes the program code, which splits the code and sends the acknowledgement back to node B. It is also possible to introduce a flow control mechanism with fraglets or other complexer programs to satisfy users requirements directly.

3.4 Map Reduce

Map Reduce operations are a common concept in functional programming languages, where a *reduce* function is applied to data transformed by a *map* function. The map operation applies the function *map* element wise to a list of elements and thus a new list is produced. The reduce operation processes a list to compute one return value. Thereby a function *reduce* is applied to the first two elements. The result and the next element in the list are combined by applying *reduce* again until the entire list is processed. Instead of the *reduce* function it is also possible to apply a $filter$ function, which eliminates those elements in a list not fulfilling a specified constraint. Another possible function is *select* to choose one specific element of a list[33].
The *map* function is applied to each element in a list independently. Therefore it is trivial to parallelize the computation. Each Compute Element of the executing machines processes one or multiple elements of the input list. This corresponds to the Data Parallelism programming pattern.
It is not possible to completely parallelize the *reduce* function. A subset of the lists can be processed in parallel, but then the combination of the intermediate results is executed sequentially by applying the *reduce* function to the intermediate results. Figure 3.1 shows a schema of the workflow.

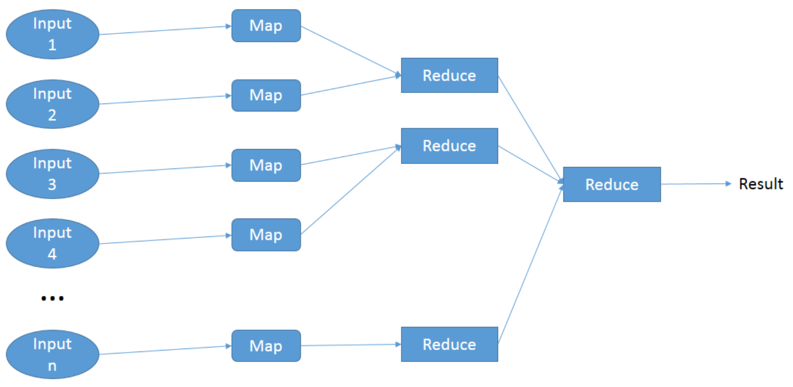


Figure 3.1: Workflow-schema of a parallel map-reduce operation. The *map* operations are independent and the *reduce* operations are applied to a subset first.

The map-reduce operation is often used for big-data procession, because it executes highly parallel and is easy to program. To employ this programming pattern one just has to define a map and a reduce function. Big Companies, e.g. Google[34], use this execution pattern in their cluster systems.

3.5 Named Functions and Cached Computations

Tschudin and Sifalakis[2] propose an extension to the existing ICN system for computations. In current ICN implementations the focus is on accessing static data, e.g. documents or images independent from their location. They point out that often users do not only want to fetch the data, but they want to use them. Moreover, sometimes it may be impractical to transfer the data to the user because of their size or even impossible for security reasons. These cases could be handled by offering names for transformations of the data. The scope of ICN is extended by NFN, where data can be modified or transformed, before they are delivered. A user can define, which functions should be applied to the data before they are delivered by using λ-calculus expressions. To express complex programs, functions written in a high level programming language can be called from within the λ-expression. By executing these λ-expressions the network is able to decide how compute and deliver the result. This means arbitrary programs can be computed in the network. In the context of NFN cloud programming and the set up of workflows is simplified and generalized.
The NFN should handle three prototypical scenarios to compute the result of computation requests. These cases are illustrated in Figure 3.2. The case *a*) avoids recomputing if a cached result is already available. This case works like a CCN request. The case *b*) will handle the case if there are no input data available and the network has to compute them. Case *c*) handles the case if the publisher of a function pinned it down on a node, either for security or copyright or for technical reasons. In this case it is required to push the computation and the data to the location where the function is pinned.

3.5.1 λ-Expressions in the Network

To resolve a λ-expression an abstract machine can be used. Tschudin and Sifalakis suggest to integrate an abstract machine in every ICN-node to handle ICN interests, which contain a

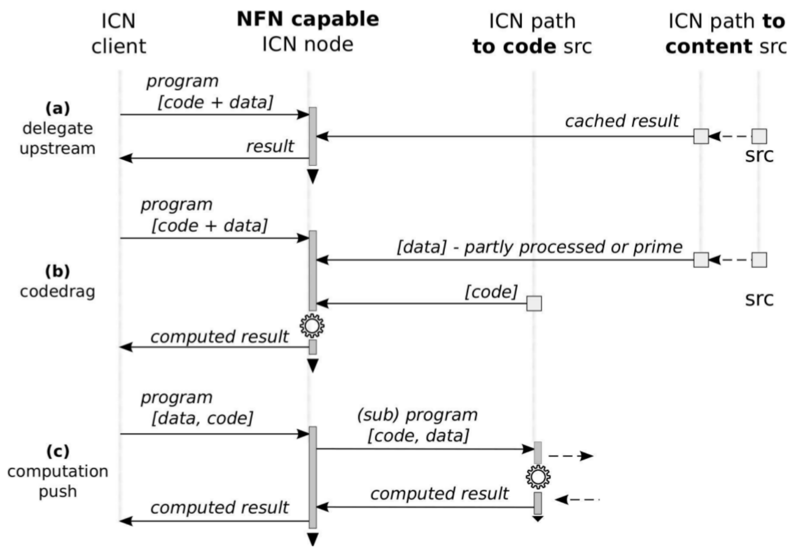


Figure 3.2: Three situations that a NFN must be able to handle[2].

λ-expression in the name. As ICN system they use the CCNx protocol described in Section 2.1.1. They propose to choose a Call-by-Name abstract machine (see Section 2.3.3), because the network system benefits from its semantics. A Call-by-Name abstract machine does not resolve a parameter before a function call is performed, but only when the parameter is required to continue the computation. Therefore, in a NFN unnecessary network traffic is avoided, if a computation cannot be finished. As abstract machine Krivine's machine is used inside a CCN-node and it is called NFN-engine from now on. To resolve a λ-expression within the network, the NFN-engine uses the existing data structures. A computation configuration of an abstract machine is the 4-tuple `<E, A, T, R>`, where `E` is the environment, `A` the argument stack, `T` the current term and `R` the result stack. The content store is used to store computation configurations of the abstract machine. As names for the configurations the hash values of the previous configuration can be used. Hashing the configuration leads to unique and deterministically computable names, so that it becomes possible to request the next configuration from the network instead of computing it.

3.5.2 Computations in the Network

It is not handy to write complex programs in λ-expressions. So it is required to enable the NFN-engine to interact with functions written in a high level programming language. These functions are accessible with their names, so that it is possible to encapsulate them in a content object and transfer them over the network. NFN uses λ-expressions to compose and organize computations and the high level functions are used to solve complex calculations and to manipulate data.

In CCN a user requests the data with the name `/path/data` by sending an interest message `ccn|path|data`, where `|` splits the different components. In NFN this is a bit different. To encode the computation `f(g(/path/data))` in an interest name the innermost component becomes the first component and the outermost component becomes the last component. The interest message becomes `ccn:nfn|/path/data|g|f`, where the `ccn:nfn` marks the interest message as NFN message. The innermost component is usually the name of the interest data. Since CCN uses the longest prefix matching pattern as forwarding strategy the interest message is routed to the input data. To compute the result the node has to request the missing functions, which are usually smaller than the input data. Thus, the network traffic is reduced.

3.5.3 The Find or Execute Instruction

A NFN-engine of a node, which received an NFN-interest message invokes the FOX (Find or Execute) instruction. The FOX instruction can be divided in three different phases shown in Figure 3.3. For example if a user wants to request the result of the computation

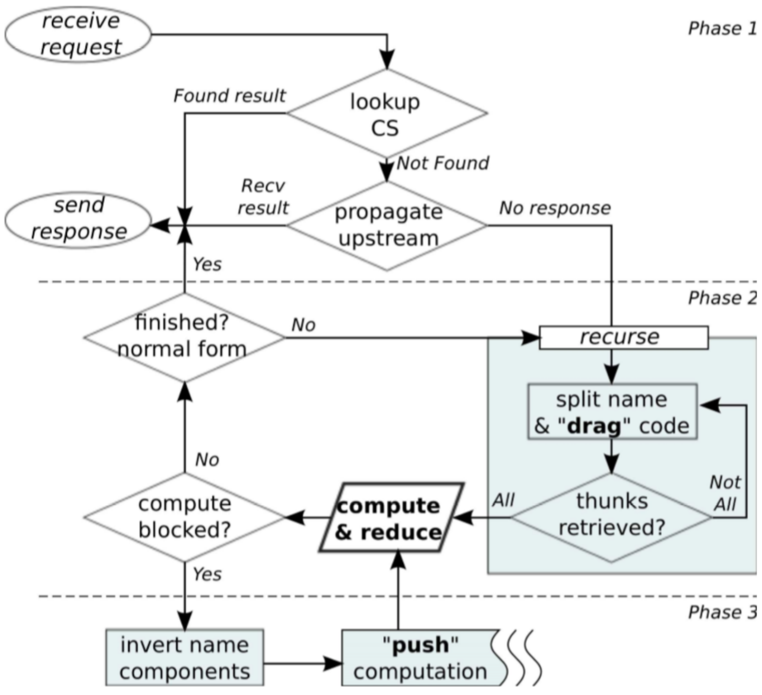


Figure 3.3: NFN combined resolution and forwarding strategy for CCN[2].

`f(g(/path/data))` he sends the reversed interest message `ccn:nfn|/path/data|g|f`.

*Phase*1 : "**Search by Upstream Delegation**"
The first phase of the FOX instruction searches if a result is available from the networks cache. Thereby the default forwarding strategy is used as in CCN. If an interest is received a node will check its local cache and if the interest cannot be satisfied it will be forwarded according to the PIT entries. Since the components of the interest are reversed, the interest is forwarded in the direction of the data source. This increases the probability to find a cached result. If there is no result available the searching process will fail and the FOX engine will enter *Phase*2.

*Phase*2 : "**Search by Upstream Delegation**"
The FOX instruction tries to compute the result by using intermediate results. In this case the rightmost component of the interest is split from the name by the FOX instruction

and interests for the split component and for the rest of the expression are issued. The rest of the expression is an intermediate result of the complete request. For the example the interest is split to `ccn:nfn|f` and `ccn:nfn|/path/data|g`. If an intermediate result and the component are found the node can combine them and deliver the result to the requester. If the result is not found the node splits further components from the name and starts searching again. If no intemediate result is found at all the FOX instruction tries to compute the result locally, thereby it is required to fetch all required resources.
To resolve λ-expressions in the network, the FOX instruction computes the hash value of the current computation configuration and uses the hash value as name to request the next configuration from the network. If there still is no result found, the FOX instruction will start computing the next configuration locally. This process works for all intermediate results and permits the network to restart a computation at any moment.

$Phase3:$ "**Push the Computation**"
If a function cannot be fetched because it is pinned to a node it is required to push the computation to this node. If a pinned function is prepended to the name of the interest message, instead of reversing the components, it will be routed to the node with this function. This node can compute the result by fetching the required resources and by using already cached computation configurations. For the example if the function `f` is pinned the interest message becomes `ccn:nfn|f|/path/data|g`

4 Named Function Networking

In 3.5 an approach is described how a CCN network can be extended to a NFN, which can deliver results of computations. We use the idea of this concept to provide an implementation and improve the system by analysing problems. To realize a NFN-system we created a NFN-node model, which is described in Section 4.1. Next we explain how the NFN-system can be improved by introducing load balancing and parallel computing. At last we give information how we integrated the NFN engine in CCN-nodes and how we encoded NFN requests in CCN-names.

4.1 NFN-Node Model

Our NFN-node model consists of three layers. The first layer is the CCN-Layer, which is responsible for the transport of messages and for communication purpose. The CCN-layer does not touch names of interests but forwards them according to the FIB entries. The second layer is the NFN-layer and is responsible for forwarding decisions of computations. The NFN-layer manipulates names of interests by reducing λ-expression and can create new interest messages, but cannot manipulate data. It is possible to perform simple computations on numbers with the NFN-layer. The third layer is the Service layer and is responsible for the manipulation of content objects taken from the underlying CCN system. The service layer is invoked by a function call of the NFN-layer. With the help of this three layer system it becomes possible to extend a CCN network with a few NFN-nodes to perform computations, whereby the input data are mainly stored on the CCN-nodes.

4.1.1 CCN-Layer

The CCN-layer is nothing more than a CCN system itself. As basis for this work CCN-lite (see Section 2.1.2) was used. The CCN-layer provides functionality to create interest- and content objects and functions to transfer them over the network. The content store of the CCN-layer is used to cache final and intermediate results of the NFN-engine, the PIT is used to administrate all running computations and the FIB to forward requests for missing resources or intermediate results.

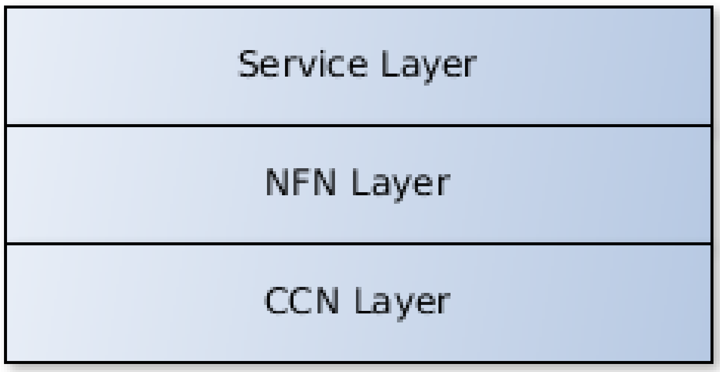


Figure 4.1: The three layer model.

4.1.2 NFN-Layer

The NFN-layer is the central part of this work. Krivine's abstract machine, which was extended with additional instructions to fit the requirements of NFN, is the NFN-engine in the NFN-layer. It is used to optimize the location where a result is computed and to find cached results or intermediate results, which can be used for the actual computation. Thereby we changed the NFN concept of Tschudin and Sifalakis[2] in some points to a more practical way. $Phase1$ and $Phase2$ of the FOX instruction are combined and $Phase3$ becomes more flexible. All three phases are integrated into a strategy system, which optimizes the location of the execution. The NFN-layer is invoked if an interest message is marked as NFN-interest and cannot be satisfied from the networks cache.

4.1.2.1 Name Manipulation

Manipulating the names of NFN-interest is the central task of the NFN-layer. By swapping the name of a NFN-interest the NFN-layer can decide where the result should be computed. Usually a NFN-interest contains more than one name. One name in the request is the name of the function and often there are other names, which specify the names of the content object, on which the function should be applied. It is also possible that more than one function should be applied. To distinguish between a content name and a function name we represent a function call explicit with the keyword `call`. To call the function `/functions/func1` with the data `/path/data` the expression `call 2 /functions/func1 /path/data` is used. The `2` describes the number of parameters that correspond to the `call`. Generalized the syntax of the `call` is: `call n` $\mathtt{p}_1$`, ..., `$\mathtt{p}_n$, with `n` $\geq$ `1`, where $\mathtt{p}_k$ is the name of the k-th parameter of the call. The parameter $\mathtt{p}_1$ is the name of the function that should be called and the parameter $\mathtt{p}_k$ with `k>1` is an arbitrary λ-expression. Unfortunately such a `call` expression is not routable by the CCN-layer, since no name is prepended. To prepend a name before an expression an abstraction (see Section 2.2) is used, so that the abstract machine can include the name by reducing the expression again. For example an abstraction can be used to extract the input data of `call 2 /functions/func1 /path/data` and the result will be (λ `x.call 2 /functions/func1 x)/path/data`. Instead of attaching the extracted name behind the λ-expression the NFN system prepends it to make it relevant for the routing process. One property of the longest prefix matching forwarding rule of CCN is to ignore the last components in an interest name if the first components match. Thus, the interest will be forwarded to the first name of the NFN-interest. Since usually data content objects have a larger size than function content objects it is beneficial to compute the result at the location where the input data are stored.

4.1.2.2 Forwarding Strategy

If a user sends a NFN-interest to a NFN-node there is usually no name prepended. If this interest contains a service layer `call`, the NFN-interest will be in the form `call n` $\mathtt{p}_1$`, ...,` $\mathtt{p}_n$, where $\mathtt{p}_1$ is thee name of the function to be called and $\mathtt{p}_k$ with `k > 1` is an arbitrary λ-expression. A NFN-node that receives a NFN-interest, which contains at least one name and has no name prepended, becomes the organizer node of the computation. The organizer node chooses a parameter $\mathtt{p}_k$ and prepends it, so that it becomes possible to forward the NFN-interest. Since this parameter should make the interest routable, only names can be chosen. In the following we will call a parameter, which is a routable name, a name-parameter.
If a node receives a NFN-interest, which contains no name-parameter i.e. no `call` it will compute the result locally.
If a NFN-node receives a NFN-interest, which has already a prepended name the node will forward it according to the entries in its FIB.
If the result is available in the content store the node can directly satisfy the interest. This corresponds to $Phase1$ in Section 3.5.3.
If there is a content object, which has a name that matches the prepended name of the interest, in the cache available the node will start computing the result locally. Thereby it is required to fetch the missing resources from the network. This corresponds to $Phase2$ in Section 3.5.3.
If the function, which should be executed is pinned on a node in the network or other input data cannot be transmitted - for example for security reasons - or the current node is a CCN-node, which has no NFN-engine the result cannot be computed and so the interest will timeout on the organizer node. In all these cases the organizer node has to choose another name-parameter to be prepended to finish the computation. If no further name-parameter is available the computation cannot be finished.
To choose a name-parameter a forwarding strategy is used. Thereby only names are considered, since an arbitrary λ-expression will not make the interest routable. The goal of the forwarding strategy is to optimize the location where a computation takes place. Thereby the forwarding strategy chooses a name-parameter to be prepended to the NFN-interest. If no result is found within a specific time interval the forwarding strategy will choose another name-parameter. It is possible to exchange the strategy of the nodes, but for consistency and to avoid routing loops it is recommended to use the same strategy on all nodes in the network. For example the forwarding strategy can use the first name-parameter $\mathtt{p}_1$ first and if no result is found it chooses the next one. This strategy will route a NFN-interest to a node, which has the function locally available first. Another forwarding strategy would be to choose the last name-parameter $\mathtt{p}_n$ first and the first name-parameter $\mathtt{p}_1$ last. This

strategy is very similar to the strategy of the FOX instruction in Section 3.5.3. The advantage of this strategy is that usually the last name-parameters are input data and the first name-parameters are functions, which should be applied. Since most often the input data objects are larger than the functions it is easier to transport the functions to the data than the data to the functions. In this work we choose this strategy. It is also possible to define more complex forwarding strategies, which choose the most explicit name-parameter (most name components) or the name-parameter, which corresponds to the largest input data first. Thereby it may be required to fetch additional information about the properties of the name-parameters (e.g. their file size of the corresponding content object) from the network.
This process is visualized in Figure 4.2.

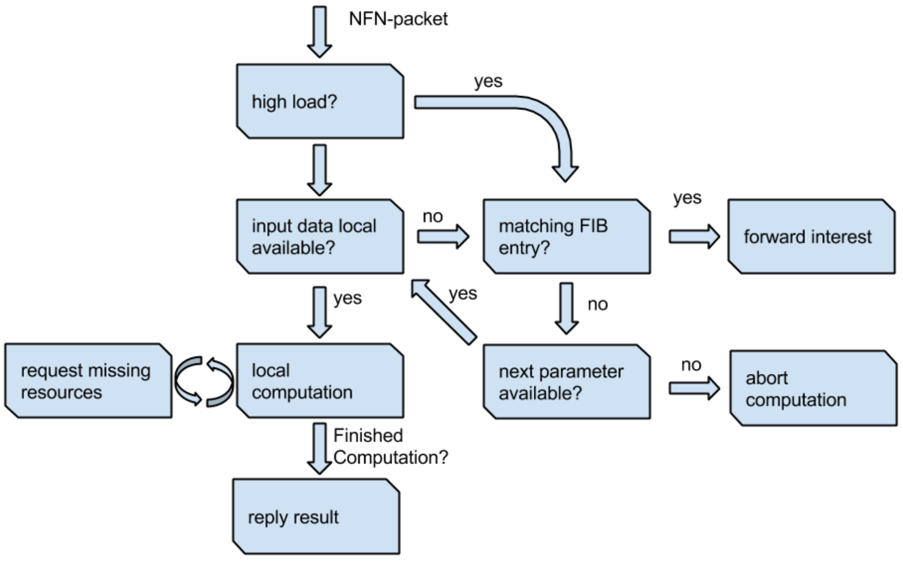


Figure 4.2: Workflow of the forwarding strategy. Depending on the current state of the network the NFN-engine decides whether it searches for a result in the network or tries to compute it locally.

4.1.3 Service Layer

The service layer is invoked by the NFN-engine if a content object should be manipulated. The core of the service layer is its engine, which executes functions written in a high level programming language. To execute functions received from the network it has also to be able to load their code dynamically. The JVM supports the concept of reflection. A Java byte code class can be loaded dynamically and executed. Therefore, we built the service layer engine on top of a JVM. Furthermore the service layer consists of an abstract machine, a PIT and a CS. The abstract machine is required to understand the computation request sent by the NFN-engine. To handle multiple requests at the same time it is necessary to introduce a PIT. Content that is required for a computation is stored in a CS.
The NFN-engine can communicate with the service layer through a face. Since we set

`/COMPUTE` as name for this face it is required to prepend the component `/COMPUTE` before an expression, so that it can forwarded to the service layer. Consequently the name `/COMPUTE` is a reserved name in the NFN network and must not be used for other purposes. A computation on the service layer appears to the NFN-engine as one computation step.
If an expression in the NFN-engine contains `call`, which marks a function call into the service layer, the NFN-engine creates a new interest message to request the result from the service layer. This interest contains the completely reduced version of the expression and has the face name `/COMPUTE` prepended.
If the service layer engine receives a computation request it fetches all required resources from the network to start the computation. Usually some of the input data are already locally available, but the function has to be fetched. If the function is pinned on another node all data have to be transmitted to that node. If all required resources are available the computation can be started. If the result is computed it is returned to the NFN-engine which continues the computation. Additionally, the service layer engine can handle call-in-call expressions. Before computing the result of the inner call expression, the service layer engine will try to find it in the network. If it is necessary to compute this result the network will optimize the location of the computation.
A complete description of the service layer can be found in Kohler's master's thesis[35].

4.2 Caching Strategies

Tschudin and Sifalakis[2] propose to use, with the cache of the CCN-nodes, an existing data structure to store computation configurations. It becomes possible to request computation configurations over the network and to benefit from intermediate results. Furthermore, a computation can be restarted at any point. As already described to store computation configurations their hash value are used as names. Unfortunately the hash values do not reflect the structure of the network and the location where to search for them. The hierarchical structure of the names of the CCN network is flatten, because a hash value has only one name component and thus, it becomes necessary to use broadcast messages to find cached computation configuration in the network. It is possible to start a computation on a node, which has the input data locally available, by using the name-parameters of the request for the routing, but as soon as we start requesting hash values the benefits of a hierarchical name structure are lost. Additionally, if a node has no matching content object to a given hash value, it cannot compute the result by itself, since hash values contain no information over the computation request. The network cannot optimize the location of a computation if it has no information over this computation. Furthermore, requesting a computation configuration from the network will require at least one Round Trip Time (RTT), if the result is already cached. If there is no cached result the node has to await a timeout, which has to be larger than the average RTT, because otherwise all requests would timeout. On a modern computer computing the next computation configuration of the abstract machine takes only a few nanoseconds, which is much faster than the average RTT. Therefore, the computation performance decreases if the computer starts requesting computation configurations from the network, since the average RTT is longer than the computation time of the next configuration. If it is required to await a timeout the problem becomes even worse.
For this reasons it is required to request only results whose computation time is longer than the time to fetch them from the network. Computation configuration are no longer stored in

the cache of the CCN-layer, but in separate data structures of the NFN-engine. We decide to cache only the results of service layer computations and call this caching strategy "service layer caching". On one hand the service layer executes functions with an arbitrary complexity on the data, and so the computation time of a service layer computation is usually large enough to benefit from requesting the result from the network. On the other hand to manipulate data the service layer has to be invoked and thus every expression, which contains a name contains a service layer invocation. Therefore, it is possible to use the real name of an interest with a meaningful prefix as name for the cached result, so that it becomes possible to find a result by using the hierarchical forwarding system of the CCN-layer. Routing NFN-interests always with the same forwarding strategy increase the probability to find a cached result. If a NFN-interest contains various service layer invocations, the entire interest could be satisfied from cache if the result was already computed. If the entire interest is not available from cache, it is possible to use the cached results of the individual service layer invocations as intermediate results to speed up the computation if they are available. The result of each service layer invocation as well as the combined result will be available from cache if the computation finished.
The following example explains how intermediate results are created. A user issues the NFN-interest `call /func1 (call /func2 (call /func3 /path/to/data))`. By prepending the input data the interest is routed to a node, which has `/path/to/data` locally available: `/path/to/data (λ x.call /func1 (call /func2 (call /func3 x)))`. First the node will check if the entire expression is available from the cache. If it is not the node will invoke the service layer to compute the result. The service layer will request the function `/func1` and check if `call /func2 (call /func3 /path/to/data)` is available from cache. To do this the name is transformed to `/path/to/data (λ x.call /func2 (call /func3 x))` to make it routable. Since it is not, the service layer starts a new computation to get the result of this subcomputation. Thereby it has to request the function `/func2` and to search for the result of `/path/to/data (λ x.call /func3 x)`. If the result is not found the service layer has to fetch `/func3`, to compute it. The input data `/path/to/data` are already locally available. The result is stored under the name `/path/to/data (λ x.call /func3 x)`. When this result is available `/func2` can be applied and the result of
`call /func2 (call /func3 /path/to/data)` is stored under the name
`/path/to/data (λ x.call /func2 (call /func3 x))`. At last the function `/func1` can be applied and the final result is cached under the name
`/path/to/data (λ x.call /func1 (call /func2 (call /func3 x)))` and replied to the user. Thus at the end of this computation the cache of the node contains:

- `/path/to/data (λ x.call /func3 x)`
- `/path/to/data (λ x.call /func2 (call /func3 x))`
- `/path/to/data (λ x.call /func1 (call /func2 (call /func3 x)))`

The intermediate results have the name prepended to make it possible to find them with an interest message, which also has name components prepended for routing purposes. The name of the input data is prepended because the computation was executed on a node, which had the input data locally available. If the forwarding strategy would route the interest to a node, which has one of the functions locally available, the name of the function would be prepended.

In the evaluation one can find a comparison between the service layer caching and the caching of every computation configuration.

4.3 Optimizations

After defining the behavior of the NFN-nodes the basic NFN-system is completed and can be employed. But to make the system more reliable and efficient in practice there are several optimizations, which can be applied. To this point only nodes next to the user can become organizer nodes. Thus, the load of these nodes is usually much higher than the load of nodes deep in the network. Therefore, it is required to introduce a load balancing mechanism.
Timeouts limit the maximal computation time. But especially computation, which are executed in the network - and NFN is designed for such computations - can have a long runtime.
Additionally, waiting on timeouts will issue a long reaction time, if a resource is not available. In the following mechanisms are discussed to avoid such problems.

4.3.1 Load Balancing

Usually the organizer node is the node next to the user. If many users are connected to the same input node this node will become the organizer for many computations. In case that the users issue a lot of computation request at the same time, which require a lot of computation power on the organizer node, it is possible that the load distribution over the network is unbalanced. This scenario can cause similar problems to a Denial of Service attack. In current computer networks a Denial of Service attack usually crashes a server or a computational resource by creating more load that can be handled. For example if many connections to a web server are created, the network infrastructure could not satisfy all requests and so some are lost. Another way to create a Denial of Service on a web server is to create computation load so that the server itself cannot handle the request. This is possible by using programming errors in the server software or by using secure connections, which require more computation power on the server side than normal connection. Since CCN is not connection based, the security is implemented on data and not on connection level and a resource can be received from any node in the network, it is more difficult to attack a CCN-node with a Denial of Service attack. For NFN this is not the case, because the user can easily issue high load on a node. High CPU load slows down the node, but high memory load is much more problematic, because the underlying memory management system kills the service if it requires more memory than available. For this reason it is required to implement a load balancing mechanism for NFN. A node has to verify that there are enough free resources before it starts a computation. If there are no free resources, a node will not accept a computation and redirect it to a neighboured node. But how can a node measures whether there are free resources? An easy way to do this is to accept only a certain number of computations. But this can be dangerous, because different computations issue different loads. Another more reliable way would be to monitor the CPU and Memory load and if one of them become critical high further computation is redirected.

4.3.2 Thunks

As already explained the organizer node sends interest messages for the computation and waits for a reply or a timeout. If a timeout occurs the organizer node will try to compute the result another way. This will be helpful, if a resource is not available or overloaded. But what if a computation timeouts because computing the result requires more time than the given timeout interval? This scenario is problematic, because nodes in the network start a computation, which requires a long time and so a lot of resources. Before the computation finishes, a timeout occurs on the organizer node and the computation is started on another node, too. Since it is very unlikely that the computation can be finished this time without a timeout, the computation is started on further nodes, but the result will not used. Thus, unnecessary load is created on the network and a result is not delivered. In the following ways around the timeout problem are proposed.
The simplest way to avoid a timeout is to increase the time before a timeout occurs. The disadvantage of this solution is that a computation can take more time than the timeout interval again. Additionally, the reaction time of the network is reduced because it waits longer before it uses another path to find a result. Therefore, a better way to handle time consuming computations is required. If a node asks for the result of a computation, it will first check if it is possible to compute the result and what the estimated computation time would be. This request is called a thunk request from now on. A thunk request is a request, if a resource is available or can be made available and is transmitted as interest message over the CCN. If a node receives a thunk request it has to verify that all required input data or functions are available. To do this a node transmits other thunk requests. If a node can deliver a content object or a result it will reply with a thunk reply message. This message is transmitted as content object. So for every thunk request either a timeout occurs or a thunk reply message is received. The thunk reply messages contain a time value, which is an estimation of how long it will take to deliver the result. Thus, the requester can adjust its timeout interval using this value.
If a NFN-request is sent to the network, which is marked as thunk request, the requester will necessarily receive a thunk reply. Afterwards another interest has to be formulated to request the result of the computation using the timeout values of the thunk. If multiple thunks are required the different timeout values need to be added to calculate the entire computation time. If no thunk reply is received within a timeout interval, the user will know that the entire result is not available without starting the computation itself. This way it is possible to save resources.

4.3.2.1 Thunks for Parallel Computing

Additionally, to the resources saving properties of thunks it is possible to enable the network for parallel computations. Because of the Church-Rosser Theorem (see Figure 2.4), which defines the confluence of the λ-calculus, the result of reduction is independent of the order of the reduction steps. This property enables the NFN system to compute results of subcomputation out of order and combine them. One can easily see that the result of an expression like `add (call /wordcount /path/to/doc1) (call /wordcount /path/to/doc2)` can be computed in parallel i.e. on different nodes, because both arguments of the add operation can be computed independently. After both partial results are available, they can be combined on one node. But how to start the subcomputations in parallel? The simplest way

to start parts of a computation in parallel is to request all resources and subcomputations directly without waiting for a reply. Thereby the same problems as before occur with time intensive computations. Thus it is required to use thunks for parallel computations, too. There are two ways to implement parallel computing with thunks. The first is to collect all thunks for all subparts. If all thunks are available, a node can directly request all subcomputations with the timeout values of the thunk. When applying parallel computations one can choose the largest thunk time value instead of adding them. The second way is even more parallel. After replying on a thunk a node does not wait for a request for the computation, but directly starts computing. If a subcomputation is not available, some nodes will compute unnecessary content, but the partial results may be reusable. The second way reduces the computation time minimal, since one round trip time before the computation starts is saved.
For distributing computation to different nodes the organizer node is responsible. When all results are available the organizer can combines them.

4.3.2.2 Branch with Thunks

Up to this point we discussed how thunks can solve the problem with long time computations and how they can be used for parallel computations. But it is required to discuss another scenario where thunks can cause problems: Branches. The problem with branches and thunks is that a thunk should discover whether it is possible to finish a computation. If the computation uses branches it will not be easy to determine whether all required resources are available, since this depends on the branch condition. For example we take a look at a simple if-else condition. A computation could look like
`ifelse (call /isavailable /doc1) /doc1 null`. As long as the function `/isavailable` can be found in the network this computation will return either the document `/doc1` or `null`. When using thunks as described in the previous section a node checks if `/isavailable` and `/doc1` are available. If the document `/doc1` is not available, no thunk will be returned, even if it would be possible to finish the computation. In this case the user should receive the result `null`. Thus, the result of the branch condition has to be computed when validating that the computation can be finished and for a branch that is not executed a thunk must not be requested. Checking the branch condition without using thunks will cause the same problems as in Section 4.3.2, if the computation requires a long time. Therefore, it is required to use thunks for `ifelse` expressions. But it is necessary to resolve the thunks directly without a specific resolve request to determine, which branch has to be chosen. Thereby it is possible, that the organizer node has to wait a longer time for a thunk reply and the thunk request may timeout. To avoid this thunk interests have to be retransmitted. This is not problematic because a thunk request does not start a computation. After solving the problems with branches the NFN-system is ready to solve general purpose and Turing complete computations in the network.

4.3.3 NACKs

After developing the NFN-system, which can solve Turing complete expressions distributed over the network we introduce a further optimization. When a resource is not available under a certain prefix, the organizer node has to wait for a timeout. Additionally, the organizer node usually is the first node in the network and thus it usually has a large hop

distance to the nodes holding the input data. Furthermore, the organizer node is a single point of failure. By adding one single feature all these problems can be avoided: NACKS. A Non-ACKnowledgement is the opposite of an ACKnowledgement. While an ACK message confirms that a packet has arrived or a resource is available, an NACK messages informs the requester that a resource is not available. The introduction of NACK messages changes the computation flow. If a node receives an interest (computation or thunk request), it will check if it can satisfy the interest or propagate it to another node. If there either is no matching entry in the FIB or there is no matching file in the CS or the answer to the request cannot be computed, the node will reply with a NACK. Thus, the previous node is informed, that there will be no answer to the request and can continue the computation without waiting for a timeout. The reaction time of the network will be enhanced. Additionally, the organizer node becomes less important. If a node receives a NACK message, it will not forward it to the organizer node directly, but will try to finish the computation itself by forwarding the interest to the next name-parameter of the strategy or by local computing. This way the control of the computation is handed over to another node in each step and the computation is executed deeper in the network.
Furthermore, the CCN-nodes can be extended to support NACKs, too. This will be beneficial, if the input data are stored on a CCN-only-node. Since the CCN-node cannot satisfy a computation request, it will reply a NACK message. CCN-nodes just forward a NACK message according to the entries in the PIT, but if a NFN-node receives the NACK it can directly start the computation next to the CCN-node, so that the input data are not transferred to the organizer node and the network load is reduced. Without NACKs the node next to the user starts a computation first and becomes the organizer node. With NACKs the organizer node is still available as fall back node if all other ways failed. The computation is started on the node next to the input data and the responsibility travels back in the direction of the organizer if it could not be solved. If the NACK message arrives at the organizer node the next name-parameter will be chosen and a new computation request will be sent in another direction. If no further name-parameter is available the organizer will try to compute the result locally and if this fails the organizer will reply with an NACK and the computation has failed.
To keep NACK messages compatible with the existing forwarding paradigm a NACK message is like a content reply message but it contains a special marker or content (e.g. `:NACK`). Therefore, it is possible to send a NACK message back over the CCN transport layer. To avoid problems with duplicated names NACK message must not to be cached.

4.4 Implementation

To implement our NFN system it was required to integrate the NFN-interests in CCN-names to transfer them over the CCN-layer and to add an annotation to mark them as NFN-interests. Additionally, it was required to invoke the NFN-engine from the CCN-layer, when a NFN-interest arrives and when a NFN-interest timeouts.

4.4.1 NFN Request Encoding in CCN Names

In CCNx protocol an interest message is defined as:

```
Interest ::= Name
```

```
            MinSuffixComponents?
            MaxSuffixComponents?
            PublisherPublicKeyDigest?
            Exclude?
            ChildSelector?
            AnswerOriginKind?
            Scope?
            InterestLifetime?
            Nonce?
            FaceID?.
```

The extensions for the NFN protocol have to be created on the top of the existing components to reach full interoperability with CCNx. The `name` of a CCNx interest message consists of components:

```
Name ::= Component*
Component ::= BLOB.
```

A computation request is integrated into the name components by using λ-expressions, so that the name of the interest contains the program code.
For example we take a look at the function code

```
(define encodeAndCompress(video)(
   compress(encode(video), zip)
 )
 encodeAndCompress(/path/to/video)
)
```

which translates to the λ-expression:

```
(λ video.(call 3 /compress (call 2 /encode video) /zip)) /path/to/video
```

where `/zip` is the compress algorithm, which is not a free parameter in this example. The numbers after call are the number of arguments that have to be transferred to the service layer if it is invoked to perform the computation. The λ-expression is integrated into CCNx name components by first choosing the last data name `/path/to/video`, so that the interest will be routed to this data. This corresponds to inverting the computation string as described in Section 4.1.2. Each component of the path `/path/to/video` is placed in one component, as it is done in CCNx. The next component is the remaining λ-expression. Only the last parameter is stored in its own component the other parameters stay in the component of the λ-expression. For the example above this leads to the interest (in xml):

```
<interest>
    <name>
         <component>path</component>
         <component>to</component>
         <component>video</component>
         <component>
             @video (call 3 /compress (call 2 /encode video) /zip))
         </component>
    </name>
</interest>
```

Instead of the λ-symbol, which is not available in ASCII the symbol @ is used. This interest will be routed to a NFN-node that has the video data `/path/to/video`. The NFN-engine of that node reduces the expression to `call 3 compress (call 2 encode /path/to/video) /zip` and prepends the `/COMPUTE` component to invoke the service layer engine. In this example the interest that invokes the service layer engine is:

```
<interest>
    <name>
         <component>COMPUTE</component>
         <component>
             call 3 compress (call 2 encode /path/to/video) /zip
         </component>
    </name>
</interest>
```

If not all program code or data are available locally on the node, the service layer will request the missing resources from the network. If all prerequisites are ready the computation can be started and the result is returned to the requester as content object with the initial name by using the CCN-layer.
If a NFN-interest has no prepended name-parameter, but e.g. starts with `call` or `add` it cannot be routed to the data source. In this case a node has to prepend a name-parameter. The node that performs this, becomes the organizer node of the computation. This process is described in Section 4.1.2.

4.4.2 NFN-Annotator

In order to designate the computation request marker fields are included into the name components to identify an interest as NFN-interest. This is important because a NFN-request has to be forwarded from the CCNx-layer to the NFN-layer. The marker field contains a name that is reserved in NFN-networks. A simple approach to do this is to integrate the marker as first name component. Unfortunately this obstructs the compatibility of pure CCN-nodes with NFN-packets because a CCN-node cannot handle the reserved name component and starts searching for the entire name. This is caused by the longest prefix matching forwarding strategy where the first component is first considered.
Putting the marker component at the end of a computation request solves this problem. The CCN forwarding strategy is based on longest prefix matching. So if the entire name does not match the last components will be ignored. For the λ-expression, which is the component before the marker, the behavior is the same and only the first components, which belong to the prepended name are considered. Therefore, a CCN-node can correctly match the name components of a NFN-interest and can forward it without modifications, since the computation and marker components will be ignored. For the NFN-interest

```
<interest>
    <name>
         <component>path</component>
         <component>to</component>
```

```
        <component>video</component>
        <component>
            @video (call 3 /compress (call 2 /encode video) /zip))
        </component>
        <component>NFN</component>
    </name>
</interest>
```

all the last two components will be ignored and three components `/path/to/video` will be used for the forwarding process.
For NFN packets the name becomes

```
Name ::= Component* Component Component
Component ::= BLOB,
```

where the first components are the data names, the second last component is the λ-expression and the last component is the NFN marker e.g. `<component>NFN</component>`. The first components disappear if no routable name is prepended. Similar to the interest messages a content object contains name components in CCNx:

```
ContentObject ::= Signature
                  Name
                  SignedInfo
                  Content.
```

The name of a content object in NFN has to be identical with the name of the interest, so that the content object can satisfy the corresponding interest.
Beside the interests and content objects requests will be required for the NFN communication, if a resource is available (thunks). A thunk message is similar to the normal interests or content objects but a thunk content object does not contain any content. A NFN-node has to distinguish if an interest is an interest for a content object or for a thunk. So another additional name component is introduced. This name component is placed before the NFN marker. Thus the last two components of a thunk are reserved:

```
Name ::= Component* Component Component Component
Component ::= BLOB,
```

where the third last component is the λ-expression, the second last component is the thunk marker e.g. `<component>THUNK</component>` and the last component is the NFN marker.
The third and last required packet type is the *NACK*. With a *NACK* message a node can reply to a request to inform the previous node that it is not possible to satisfy a certain interest. By transmitting *NACK* messages timeouts can be avoided on NFN-nodes as well as on CCN-nodes can be avoided. A *NACK* message is a content object, which is replied to an interest. Thus it is possible to use the existing PIT entries to forward the content object. A *NACK* message either has a special marker field or contains a special content so that the other nodes can understand it. In our case we use the content `:NACK` for the *NACK* messages:

```
ContentObject ::= Signature
                  Name
```

`SignedInfo`

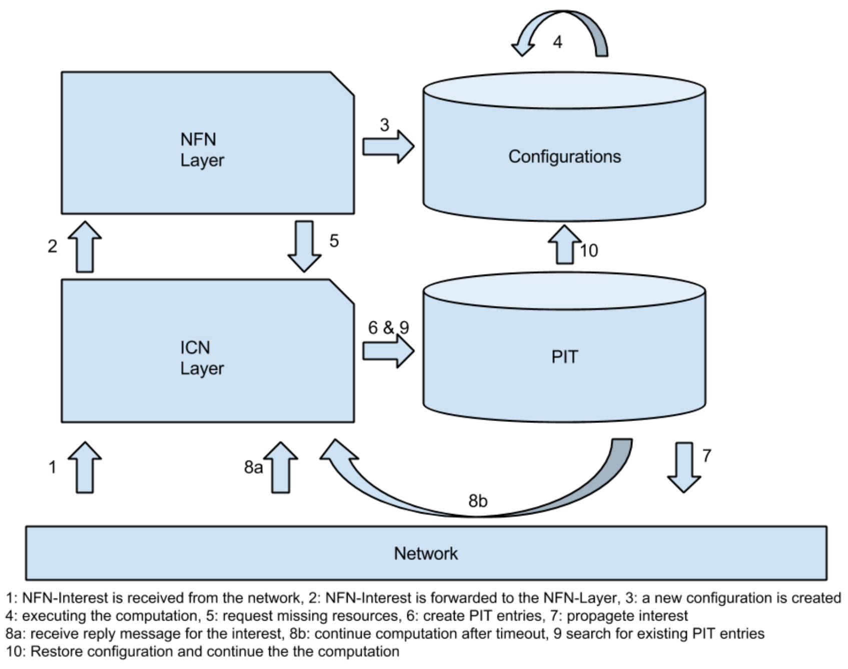
`":NACK".`

In Table 4.1 the additional name components for the NFN protocol are summarized.

Table 4.1: Additional name components for the NFN protocol

Message type	Additional name components
NFN-interest	Last name component is `<component>NFN</component>`
NFN-content object	Last name component is `<component>NFN</component>`
NFN-thunk interest	Last name components are `<component>THUNK</component> <component>NFN</component>`
NFN-thunk content object	Last name components are `<component>THUNK</component> <component>NFN</component>`
NACK	No special name component, but `:NACK` as content

4.4.3 Integrating an Abstract Machine in a CCN-Node



1: NFN-Interest is received from the network, 2: NFN-Interest is forwarded to the NFN-Layer, 3: a new configuration is created
4: executing the computation, 5: request missing resources, 6: create PIT entries, 7: propagete interest
8a: receive reply message for the interest, 8b: continue computation after timeout, 9 search for existing PIT entries
10: Restore configuration and continue the the computation

Figure 4.3: Integrating a NFN-engine into a CCN-lite node. If an interest is marked as NFN-interest it is forwarded to the NFN-engine, which may causes further interest. If a computation finished a content object is replied as answer and the configuration is removed.

There are several points where the NFN-engine has to interact with the CCN-node. A NFN extended node handles an CCN interest on the same way as an CCN-only-node does, but a NFN-node has to check whether there is a NFN marker in the interest (Step 1 in Figure 4.3). If it is found the NFN-node has to forward the interest to the NFN-engine, which decides how to process the computation (Step 2 in Figure 4.3). The NFN-engine can cause further interest messages to forward the computation or to fetch intermediate results. These interests are propagated like normal interests by using the FIB of the CCN system (Step

$5, 6, 7$ in Figure 4.3). Interests for computations are added to the PIT. On a CCN-node a PIT entry maps the name of a pending interest to a face (in fact to the id of the face), on which the interest was received. The node will forward the answer to the face with this id. Assume there is a reserved range of face ids for the NFN system. If a NFN-node adds an interest to the PIT, it will choose an id out of this range. If the node receives the reply to this interest it will find a matching entry in the PIT with a face id in the NFN range (Step $8a, 9$ in Figure 4.3). Thus it does not forward the content to the next node, but it invokes the NFN-engine to continue the computation with the new input data (Step 10 in Figure 4.3). While waiting for the reply of an interest the current computation is frozen, so that the node can handle other interests and computations. To freeze a computation the entire computation configuration (4-tuple: `<E,A,T,R>`, see Section 2.3) is stored in a data structure of the NFN-engine, so that multiple configurations can be handled at the same time. As data structure a list can be used. A configuration in the list can be accessed by the face id of the computation. Beside the current configuration of the abstract machine it is required to store the name of the original interest, the face id and timeout information in the data structure.
While a CCN-node removes an interest on a timeout, for computations a NFN-node must not remove the interest, but the NFN-engine has to be restarted the computation and to decide how to continue (Step $8b$ in Figure 4.3). To make this decision a strategy is applied (see Section 4.1.2).

5 Testing and Evaluation

In this section different testing scenarios for the NFN-engine are described. The scenarios are tested on native computers and in an OMNeT++ simulation. As testing scenarios toy applications are used to show the functionally of the implementation. A NFN is freely programmable and one can run arbitrary applications within it. In the following the two different caching strategies described in Section 4.2 are tested and compared. Furthermore, the optimizations from Section 4.3 are evaluated and the behaviour of the network with and without these optimizations is compared. All tests are run five times. To compare the performance between the different test runs the mean of the required time of all tests is used. All experiments are executed within a test environment on a single machine. For the tests a Apple Macbook Pro Retina First Generation is used.

5.1 Performance of Different Caching Strategies

In Section 4.2 we introduced two different strategies how intermediate results can be cached. The first strategy consists of caching every computation configuration by using the hash value as name for the cache entry whereas the second strategy caches only service layer operations. In a first step of the evaluation these two strategies are compared. A test scenario with two nodes N1 and N2 is given. Both nodes are connected with a face. A user always sends his interests to node N1 first. All tests are executed for to different scenarios. In the first run there is no intermediate result available in the network, in the second run all required intermediate results are stored on the second node $N2$. As simple sample computation `add 1 2` is used. Our abstract machine has to compute 89 machine configurations to compute the result. In the first execution, when no intermediate result is available, $N1$ asks $N2$ for every computation step if the next result is available. Since there is no intermediate result available every request times out and $N1$ has to execute them locally as shown in Figure 5.1. The execution time increases for the entire computation. Therefore, an average time of 32.15 seconds is required for a simple addition. If the computation configurations are already available from the caches of the node $N2$, the computation time will become lower, since it is not necessary to await timeouts but only to transmit the computation configurations from $N2$ to $N1$., shown in Figure 5.2 The computation time for

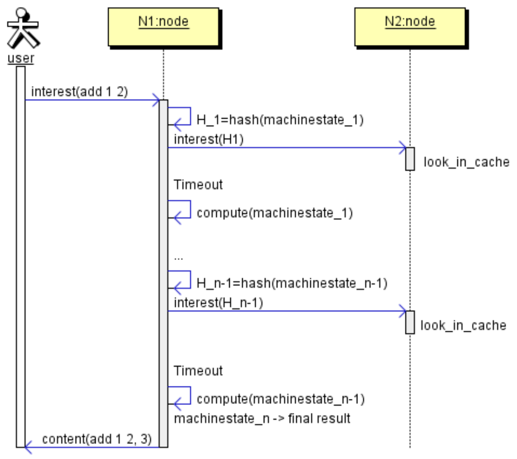


Figure 5.1: Caching and requesting every computation configuration. Before starting a local computation the NFN-engine has to await a timeout.

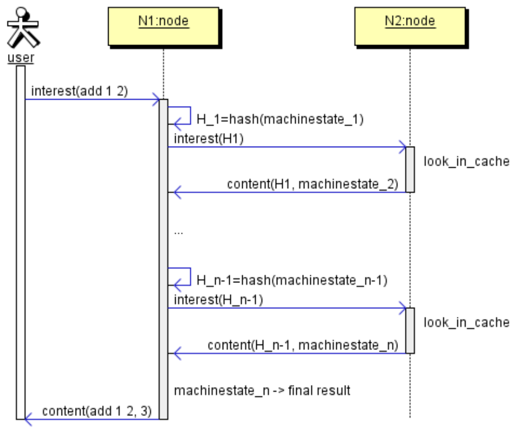


Figure 5.2: Caching and requesting every computation configuration. No local computation required since all computation configurations are available on node $N2$.

the simple addition was 2.25 seconds on average. Nevertheless, the execution time for this example is very high, an addition can be computed within nanoseconds on a local machine. If the same example computation is executed within an NFN-engine, which only uses the service layer caching, a simple addition will be executed at the first node only, since the addition contains no routable name. A user sends an interest to the node $N1$, which performs the computation even if there is a cached result on node $N2$. Since the computation time is negligibly low on node $N1$ the computation time is about the Round Trip Time (RTT) between the user and $N1$. Thus, the result can be computed within 0.08 seconds, which is much faster than in the previous test. The test result is the same for both runs, since the node $N1$ does not request any content from cache and so it does not matter if node $N2$ has a cached result or not. The results are summarized in table 5.1.

In Section 4.2 it was already pointed out that caching every computation configuration has a too fine granularity. This is because the RRT between two nodes is higher than the time, which is required to compute the next computation configuration. So it becomes impossible to speed up a computation by receiving a computation configuration from the network. The

Table 5.1: Evaluation results of the different caching strategies for a simple addition. The time values are the average values over five runs.

Tests	Caching of configurations	Caching of service layer results
No cached results	32.15	0.08
Cached results on node $N2$	2.25	0.08

granularity has to be increased, so that only intermediate results are requested, which can be transmitted over the network more quickly than computed. Besides the computation time should be large compared with the timeout intervals to minimize the overhead if there is no intermediate result found in the network. Using the service layer caching NFN-engine simple operations on numbers are executed locally by a node. Thus the communication overhead is removed for simple operations. Test scenarios how the service layer caching works on routable names can be found in Section 5.2. Since caching all computation configurations is extremely inefficient and slow in the following we concentrate on a NFN-engine, which only caches the results of service layer operations.

5.2 Service Layer Caching and Optimization

In this Section we describe the evaluation of the service layer caching system (Section 4.2). We start with explaining the behavior of the NFN-engine by showing different scenarios and how the network can handle exceptions. For this different test scenarios were developed on the same topology. In Section 4.3 there is a description how the performance of the NFN-engine could be improved. The different optimizations are compared to show their performance improvements. We start with enabling thunks to perform operations, which time out on a system without. Next we will demonstrate the performance improvements if thunks are used for parallel computing. At last we take a closer look at the behavior of the network when NACKS are enabled to show the faster reaction time.

5.2.1 The Test Environment

For the evaluation a simple test environment is used, on which different scenarios can be performed and, on which can be evaluated how the network reacts. The test environment consists of five nodes. Figure 5.3 shows the topology. Two of the nodes are CCN-lite nodes and the remaining three nodes are NFN-nodes based on the CCN-lite system. The NFN-nodes are connected with a service layer, which is implemented in Scala and runs Java byte code. All nodes are simulated on one computer. As test machine a Apple Macbook Pro Retina First Generation is used. On each node a document is stored. The prefix of the document is `/nodeX`, where X is the number of the node. Each document contains a specific number of words. Document X contains X words, so that for example the document on the node with the prefix `/node3` is a document with 3 words in it. In the following we call the node with the prefix `X: Node_X`. The test program `wrdcnt` counts the words within the documents so it becomes easy to verify, whether the result is correct. To simulate a more complex program a timeout of 500ms is added to the program. For the evaluation the function `/bin/scala/wrdcnt` is added to $Node_3$ and $Node_4$. Since no routing algorithm is used for the test examples the FIB is initialized manually and forwarding rules for each prefix are installed in a way that every node can find every content. Additionally for each node

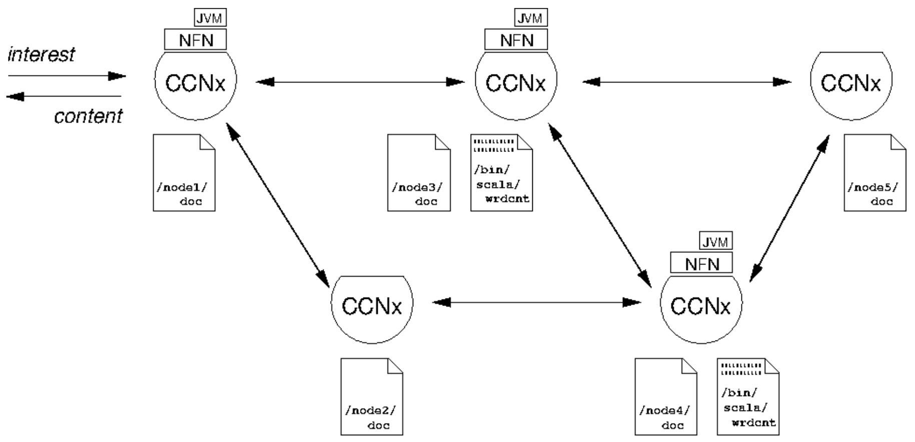


Figure 5.3: The topology of the test environment. Three nodes are NFN-node, two nodes are CCN-lite-nodes.

forwarding rules to the function `/bin/scala/wrdcnt` are installed. The user that inserts the test programs into the test environment is connected with $Node_1$. Thus $Node_1$ will be the organizer node for the computations.

5.2.2 Experiments

In this section different experiments and their results are shown. We start with simple experiments that confirm the correctness of our NFN implementation according to the strategies shown in Section 4. Later the performance of the optimization proposed in Section 4.3 are compared with the basis implementation. For the experiments the topology and the test environment described in Section 5.2.1 are used. In some experiments changes on the environment are applied to show the flexibility of NFN. The changes are described for each experiment. After each experiment the environment is reset and all removed links are restored and all cached content objects and results are deleted.

5.2.2.1 Experiment 1: Fetching Binary Code from the Network

The first example shows the functionality of the network to fetch code from a node. We use as simple computation a function call on the service layer, which counts the words in a document. In this example the words in document `/node1/doc` shall be counted. For this purpose the program i_1 = `call /bin/scala/wrdcnt /node1/doc` is used. The user sends this program to the node 1, which becomes the organizer node. First the organizer node transforms the interest to route it to the input data. The transformed interest is: `/node1/doc(` λ `x0.call /bin/scala/wrdcnt x0)`. For all test experiments the name-parameter is chosen by the strategy explained in Section 4.1.2, which chooses the last name-parameter first and the first name-parameter last. On $Node_1$ the input data are already available. Thus the computation is locally started on the service layer of $Node_1$. But on this node the wordcount function is not stored on $Node_1$, so that it is required to fetch the function from the network. An interest (i_2) is sent to $Node_3$ and $Node_4$ or to one of these nodes (this depends on the CCN forwarding strategy). The nodes reply with a content object to transfer the function. After receiving and loading the function the service layer of $Node_1$ can compute the result and reply it to the user. The result of this computation is 1. The work flow of this example is shown in Figure 5.4.

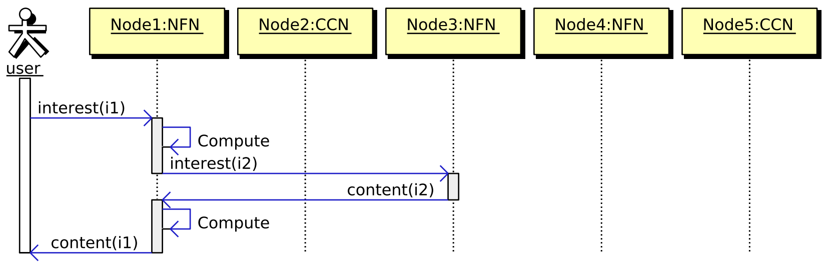


Figure 5.4: Experiment 1: Fetching a function from the network to compute a result
i_1`{call /bin/scala/wrdcnt /node1/doc}`
i_2`{/bin/scala/wrdcnt}`

5.2.2.2 Experiment 2: Push a Computation

The second example shows how a node can handle the case, if the result cannot be computed on the node that stores the input data. This scenario will be possible if the function is pinned or if the input data are stored on an CCN-only-node. In this example we choose the case where the input data are stored on an CCN only node. For our test environment we use the program i_1 = `call /bin/scala/wrdcnt /node5/doc`. The user sends this program to the $Node_1$. Since the input data are not available on $Node_1$, the node has to transform the interest to route it to the input data. Thereby it prepends the input data by applying and abstraction. The program is transformed to i_2 = `/node5/doc` (λ `x0.call /bin/scala/wrdcnt x0`), so that it can be routed to node 5 where the input document is stored. Unfortunately $Node_5$ is an CCN-only-node and thus the interest will timeout. According to the routing strategy $Node_1$ prepends the next name, which is the function name and creates the program i_3 = `/bin/scala/wrdcnt` (λ `x0.call x0 /node5/doc`). This program is routed to $Node_3$ (or $Node_4$ depending on the CCN forwarding strategy). $Node_3$ has the function locally available and therefore it can invoke the service layer to compute the result. To do this the service layer has to fetch the input data from $Node_5$ (i_4). After computing the result it is replied to the user. The result of this computation is 5. Figure 5.5 shows the workflow.

5.2.2.3 Experiment 3: Relocation of the Computation if a Link Fails

In this experiment the same computation (i_1) as for the experiment 2 is used, but the environment is changed. The link between $Node_1$ and $Node_3$ is removed. Because of this the network cannot apply the same way as in experiment 2 to complete the computation. Similar to before the network starts by trying to compute the result on $Node_5$. For this purpose the interest (i_2) is now routed over $Node_2$. But it is still not possible to compute the result on $Node_5$, since this node is an CCN-only-node. After the timeout on the organizer $Node_1$, it modifies the interest to compute the result on the location where the function is stored (i_3). This interest is routed to $Node_4$ because the connection from $Node_1$ to $Node_3$ was cut. Since $Node_4$ has the function locally available the computation is now started on $Node_4$. Therefore by modifying the network NFNs will not just simply reroute the computation to compute the result, but it is also possible that another node computes the

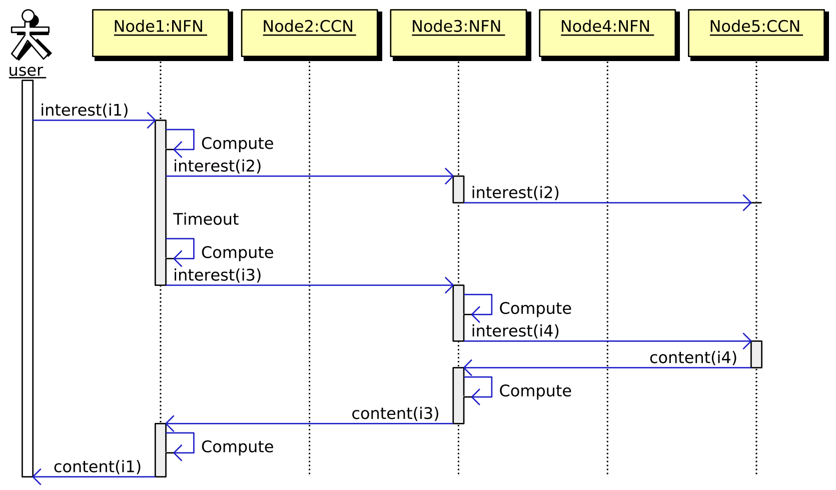


Figure 5.5: Experiment 2: Pushing a computation if it is not possible to compute the result on the node, which stores the input data.
i_1{`call /bin/scala/wrdcnt /node5/doc`}
i_2{`/node5/doc` (λ `x0.call /bin/scala/wrdcnt x0`)}
i_3{`/bin/scala/wrdcnt` (λ `x0.call x0 /node5/doc`)}
i_4{`/node5/doc`}

result. This property gives the network more flexibility and makes it more robust. Figure 5.6 shows the workflow. One can see that the workflow is very similar to experiment 2 shown in Figure 5.5, but the final result is computed on another node.

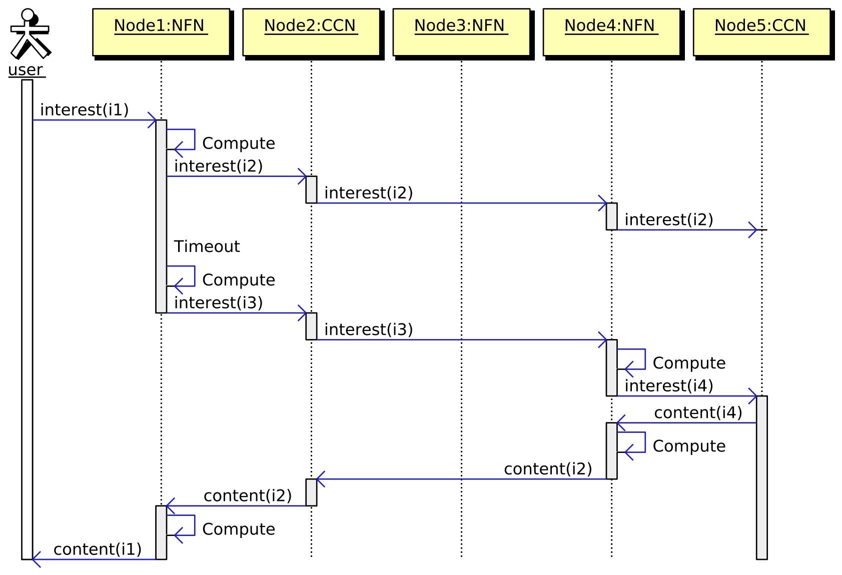


Figure 5.6: Experiment 3: The same computation as in experiment 2, but the link between $Node_1$ and $Node_3$ is cut. Thus the network finds another way to deliver the result.
i_1{`call /bin/scala/wrdcnt /node5/doc`}
i_2{`/node5/doc` (λ `x0.call /bin/scala/wrdcnt x0`)}
i_3{`/bin/scala/wrdcnt` (λ `x0.call x0 /node5/doc`)}
i_4{`/node5/doc`}

5.2.2.4 Experiment 4: Simple Map Reduce - Recursive Distribution

Experiment 4 shows how a simple Map Reduce operation can be implemented in a NFN. As `map` function the wordcount function is chosen. The `reduce` function is an addition, which is already integrated as basis operation in the NFN-engine. In this example the words in two different documents are counted and added together. The computation for this example is $i_1 =$ `add ((call /bin/scala/wrdcnt /node3/doc) (call /bin/scala/wrdcnt /node4/doc))`. After $Node_1$ received the computation is starts evaluating it. The NFN-engine on $Node_1$ splits the computation and the results of the subparts are computed on different nodes. This experiment shows that the NFN can recursively break down complex expressions to subexpressions and distribute them over the network. Thereby the network can optimize where to evaluate a subexpression. The computation is split into two subcomputations, which are distributed to $Node_3$ and $Node_4$ according to the input data. The first subcomputation is $i_2 =$ `/node3/doc (`λ` x0.call /bin/scala/wrdcnt x0)` and the second subcomputation is $i_3 =$ `/node4/doc (`λ` x0.call /bin/scala/wrdcnt x0)`. The first subcomputation is transfered to $Node_3$. After receiving the result of the first subcomputation from $Node_3$ the second subcomputation is executed on $Node_4$. If both results are received by $Node_1$, it can apply the add operation and combine both results before the final result is replied to the user.
The program flow is shown in Figure 5.7. The result of the computation is 7. It is also possible to perform this computation in parallel. This is shown in Section 5.2.2.7.

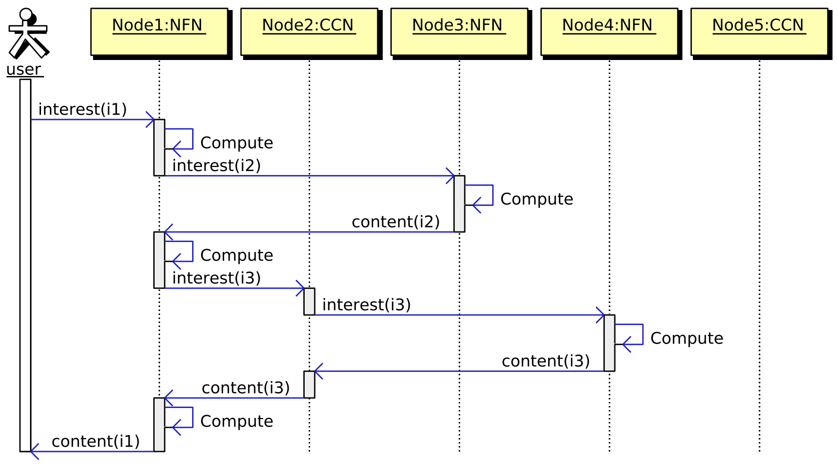


Figure 5.7: Experiment 4: sequential evaluation of sub-expressions
i_1{`add ((call /bin/scala/wrdcnt /node3/doc) (call /bin/scala/wrdcnt /node4/doc))`}
i_2{ `/node3/doc (`λ` x0.call /bin/scala/wrdcnt x0)`}
i_3{ `/node4/doc (`λ` x0.call /bin/scala/wrdcnt x0)`}

It is also possible to define more complex reduce functions by using the basic primitives, since a user can define its own functions in the λ-calculus. For example it is possible to define a function to add three result of a `map` function: `let add3 =` $\lambda x.\lambda y.\lambda z.$`add (add x y) z endlet`. This way it is possible to define an arbitrary reduce function. Furthermore it is possible to define a reduce function by using the service layer.

5.2.2.5 Experiment 5: Caching to Avoid Recomputation

The NFN does not only optimize the location where a computation takes place, but it also avoids recomputations by using the CCN cache. In this experiment we show the functionality of the cache system of NFN. We start with a simple computation. The result of this computation is cached. Later a more complex operation is executed. Thereby the simple computation is a subcomputation of the more complex one. This part of the computation is only executed once. For this example we start with the computation i_1 = `call /bin/scala/wrdcnt /node3/doc`. $Node_1$ transforms this computation to
i_2 = `/node3/doc (λ x0.call /bin/scala/wrdcnt x0)` and forwards it to $Node_3$, which performs the computation. The result is replied to the user. On $Node_1$ and $Node_3$ the result is cached. If the user issues another computation
i_3 = `add ((call /bin/scala/wrdcnt /node3/doc) (call /bin/scala/wrdcnt /node4/doc))`
such as in experiment 4, only one part of the computation has to be resolved. Since the result of i_2 = `/node3/doc (λ x0.call /bin/scala/wrdcnt x0)` is already in cache only the interest i_4 = `/node4/doc (λ x0.call /bin/scala/wrdcnt x0)` is transmitted. After receiving the result from $Node_4$ and taking the cached result $Node_1$ can compute the final result and reply it to the user. This way the cached result is reused and it is not required to compute this part of the expression again. Figure 5.8 shows this process. The result of the first computation is 3, the result of the second computation is 7.

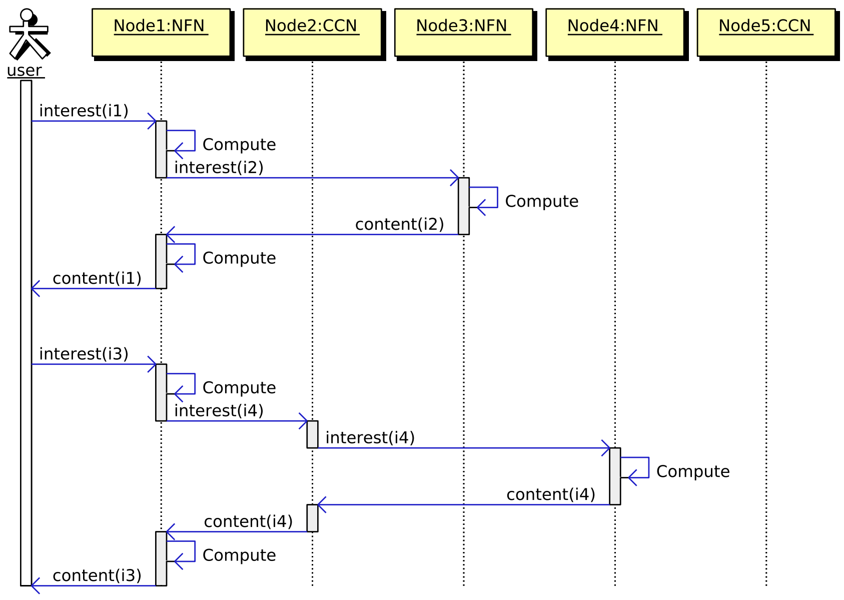


Figure 5.8: Experiment 5: caching of intermediate results
i_1{`call /bin/scala/wrdcnt /node3/doc`}
i_2{`/node3/doc (λ x0.call /bin/scala/wrdcnt x0)`}
i_3{`add ((call /bin/scala/wrdcnt /node3/doc) (call /bin/scala/wrdcnt /node4/doc))`}
i_4{`/node4/doc (λ x0.call /bin/scala/wrdcnt x0)`}

5.2.2.6 Experiment 6: Load Balancing

This experiment demonstrates the load balancing mechanism of NFN (Section 4.3.1). It can happen that a node receives many requests for computation, which should be executed locally according to the NFN policies. In this case it is possible that load balancing becomes necessary. A node either measures its load by measuring the CPU or Memory load or it simple counts the number of running computations. If there is no remaining capacity for another a node will not accept further computations to avoid crashes. In this case it is required to forward the computation to a node in the neighborhood. Assume that $Node_3$ is under high load. Additionally, the link between $Node_4$ and $Node_5$ was cut. If a user adds another computation $i_1 =$ `call /bin/scala/wrdcnt /node5/doc` $Node_1$ will transform it to $i_2 =$ `/node5/doc` (λ `x0.call /bin/scala/wrdcnt x0`) and will forward it to $Node_5$ via $Node_3$. Since $Node_5$ is an CCN-only-node the interest timeouts and $Node_1$ handles the timeout by transmitting the interest $i_3 =$ `/bin/scala/wrdcnt` (λ `x0.call x0 /node5/doc`) to $Node_3$. Unfortunately $Node_3$ cannot handle the interest even if it should according to the NFN strategy, because it is under high load. Thus it will forward the interest to $Node_4$ according to the FIB. This node can handle the computation and computes the result by requesting the input data (i_4). This interest is routed over $Node_3$ because the connection between $Node_4$ and $Node_5$ was cut. The final result is 5 and it will be replied to the user.

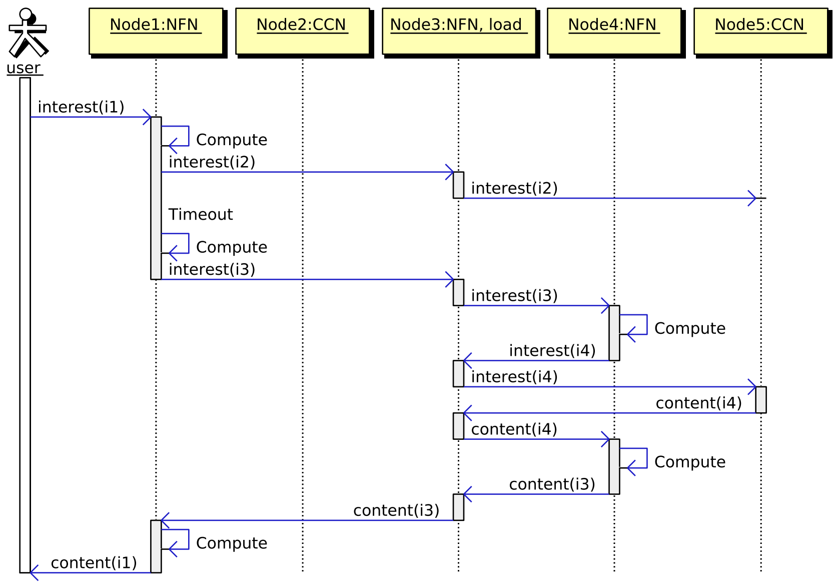


Figure 5.9: Experiment 6: Load balancing
i_1\{`call /bin/scala/wrdcnt /node5/doc`\}
i_2\{`/node5/doc (`λ` x0.call /bin/scala/wrdcnt x0)`\}
i_3\{`/bin/scala/wrdcnt (`λ` x0.call x0 /node5/doc)`\}
i_4\{`/node5/doc`\}

5.2.2.7 Experiment 7: Parallel Computing

This example presents the advantage of the thunks. With the help of thunks it is possible to perform operations in parallel. Using a Map-Reduce like operation is beneficial to demonstrate the parallel computation. In Section 4.3.2.1 one finds a description how thunks can be used to apply parallel computing. Thunks will be used to verify if it is possible to complete a computation. But after notifying the requesting node that it is possible to compute the result, the computation is directly started. Therefore one node receives the thunk and can

continue the computation while the other nodes prepare the result for the thunk. To demonstrate this the same computation as in experiment 4 is used. The user sends the interest i_1 = `add ((call /bin/scala/wrdcnt /node3/doc) (call /bin/scala/wrdcnt /node4/doc))` to $Node_1$. A user requests a thunk for this computation. To check if the computation is computalbe, the computation will be split into the subcomputations
i_2 = `/node3/doc (λ x0.call /bin/scala/wrdcnt x0)` and
i_3 = `/node4/doc (λ x0.call /bin/scala/wrdcnt x0)`. First $Node_1$ requests a thunk for i_2 from $Node_3$. Since $Node_3$ can compute the result it replies with a thunk and starts the computation directly. $Node_1$ receives a thunk reply for i_2, thus it can continue the computation and request i_3 from $Node_4$ while $Node_3$ is still computing. $Node_4$ also replies with a thunk and starts the computation. Thus both computations are running at the same time and the result is computed in parallel. To maximize the parallelism a node will resolve a thunk only if it needs the result for the next step. To add both subresults $Node_1$ has to request them from $Node_3$ and $Node_4$ by sending the same interests as before without the thunk component. Since the thunk reply contains the expected computation time $Node_1$ can adjust its timeout interval. The workflow of this example is shown in Figure 5.10.

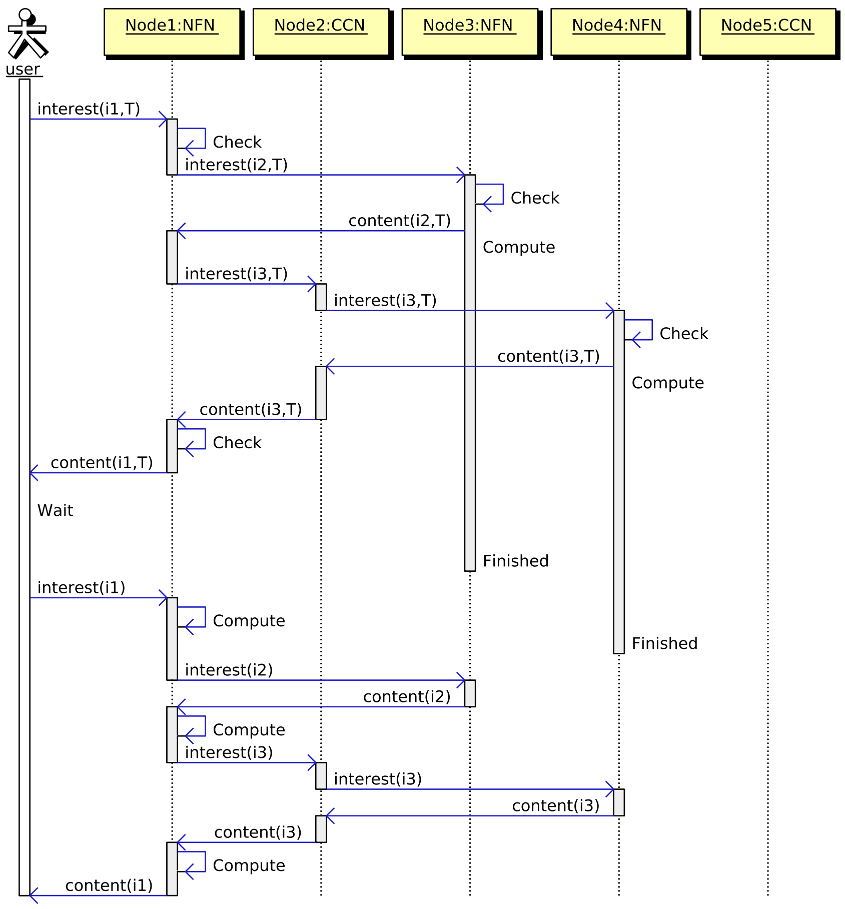


Figure 5.10: Experiment 7: parallel evaluation of sub-expressions
i_1{`add ((call /bin/scala/wrdcnt /node3/doc) (call /bin/scala/wrdcnt /node4/doc))`}
i_2{ `/node3/doc (λ x0.call /bin/scala/wrdcnt x0)`}
i_3{ `/node4/doc (λ x0.call /bin/scala/wrdcnt x0)`}

5.2.2.8 Experiment 8: Regaining Partial Control of Opportunism

Sometimes a user wants to influence the routing because he has knowledge, which is beneficial for the routing and the network does not have it. Since an interest with a prepended name is not touched by a NFN-node unless it has the content object to the prepended name locally available, this property can be used by the user to determine where to route an interest message. Therefore, a user can force the network to route an interest to specific input data. This will be a big benefit if the input data are very large but the result which is computed out of the input data is very small. This example will demonstrate the NFN system if a user sends a request with already prepended name components. For this experiment the function `wordcount` is removed from $Node_3$. A user sends an interest $i_1 =$ `/node3/doc (`λ `x0.call /bin/scala/wrdcnt x0` to $Node_1$. The node does not touch the interest since there is a prepended name, to which it has no matching content. According to the FIB the interest is forwarded to $Node_3$, which can compute the result after fetching the function (i_2) from $Node_4$.
It is also possible that the user prepends the function:
$i_3 =$ `/bin/scala/wrdcnt (`λ `x0.call x0` $/node3/doc!$. In this case the interest is forwarded to $Node_4$, which is the only node in the network, which has the function local available. Since the input document `/node3/doc` is not stored on $Node_4$ it has to be fetched from the network (i_4). This process is shown in Figure 5.11

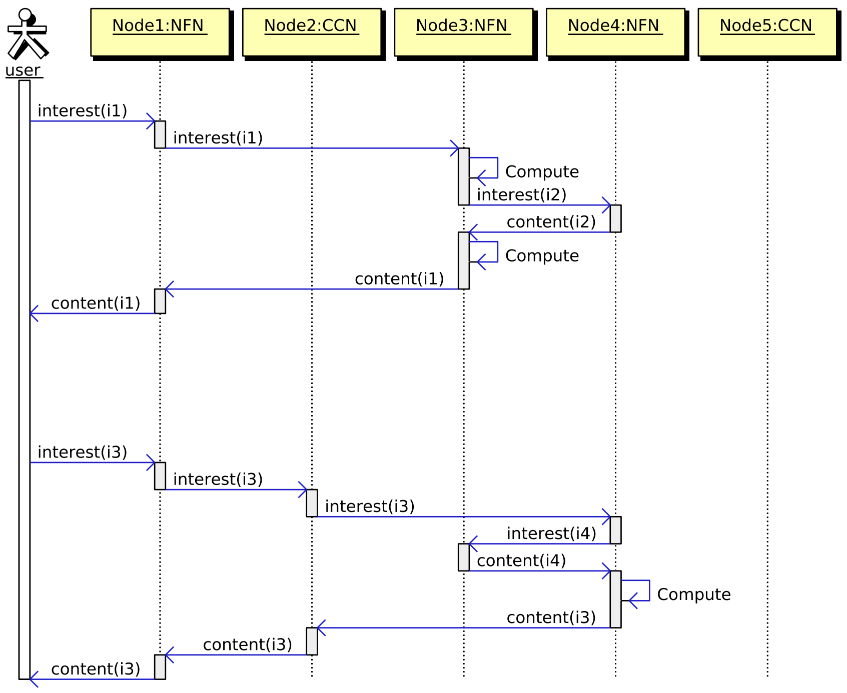


Figure 5.11: Experiment 8: A user can influence the routing
i_1{`/node3/doc (`λ `x0.call /bin/scala/wrdcnt x0)`}
i_2{`/bin/scala/wrdcnt`}
i_3{`/bin/scala/wrdcnt (`λ `x0.call x0 /node3/doc)`}
i_4{`/node3/doc`}

5.2.2.9 Experiment 9: NACKs

This experiment demonstrates the advantage of NACKs. The big advantage of NACKs is not only that they reduce the reaction time of the network, they also give the possibility to avoid decisions and computing on the organizer node and move the computation deeper in the network. One can find a description of NACKs in Section 4.3.3. A user issues the same interest as in experiment 2: $i_1 =$ `call /bin/scala/wrdcnt /node5/doc` and transmits it to $Node_1$. $Node_1$ transforms the interest to $i_2 =$ `/node5/doc (λ x0.call /bin/scala/wrdcnt x0)`. This interest is transmitted to $Node_5$, but it cannot satisfy the interest because it is an CCN-only-node. Thus $Node_5$ replies with a NACK. When $Node_3$ receives this NACK it checks whether it can compute the result. Since the function is locally available, $Node_3$ can directly start the computation without involving organizer $Node_1$. Therefore it is not required to await a timeout on $Node_1$, the network reacts faster to the fact that $Node_5$ is an CCN-only-node and relocates the computation to $Node_1$. If for some reasons the computation fails on $Node_3$ the organizer will hook in. This workflow is summarized in Figure 5.12

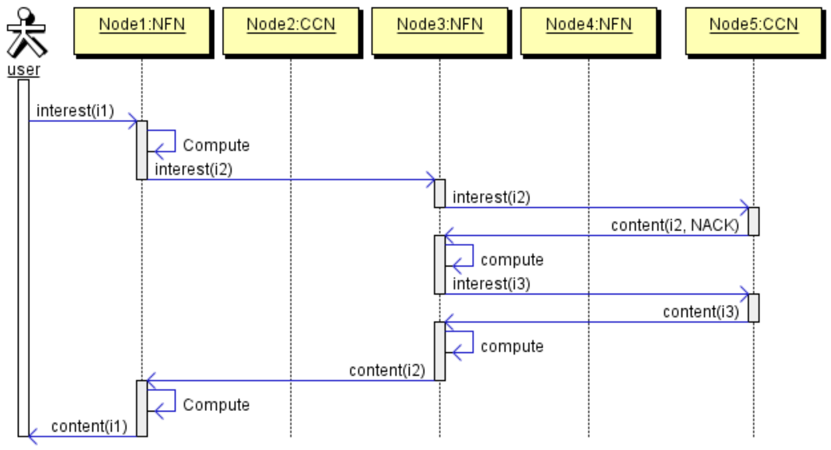


Figure 5.12: Experiment 9: NACKs
i_1{`call /bin/scala/wrdcnt /node5/doc`}
i_2{`/node5/doc (λ x0.call /bin/scala/wrdcnt /node5/doc)`}
i_3{`/node5/doc`}

5.2.3 Comparing the Results

In this Section the results of our experiments are summarized. We start with comparing the runtime of the different examples. This is especially interesting for experiments 2 and 9 as well as for experiment 4, 5 and 7, because the same computation was run with different settings. In experiment 9 the computation of experiment 2 was executed with NACKs enabled. Since NACKs avoid the timeout at the organizer node we expect a faster computation in experiment 9. This is confirmed by the measurements shown in Table 5.2. Figure 5.13 shows the timeouts for the different experiments. Thereby one can see that in experiment 2 there are three timeouts. The timeout occurs because $Node_5$ is an CCN-only-node and is the only node with the input data locally available. The fact that three timeouts occur is caused by the CCN protocol, which retransmits an interest twice before it finally times out.
Even if experiment 5b is the same computation as used in experiment 4 it is executed much

faster. This is because only one part of the subcomputations is executed, since the other part is available locally from cache (computed by experiment 5a). Thus nearly half the computation time is saved and only one of two nodes has to compute a result. Experiment 7 is the parallelised version of experiment 4. One can see that if the computation is executed on two nodes in parallel it is nearly twice as fast as the computation executed sequentially. Executing a single subcomputation takes around 500 ms. If both subcomputations are executed serially the program needs more than 1000 ms. Using parallel execution the time drops under 600 ms, which is less than the accumulated execution time of the subcomputation. This fact proofs that the subcomputation are concurrently executed. Experiment 5 was split into two parts. The first part is the small subcomputation, the second part is the entire computation.
Table 5.2 shows the time measurements of the experiments.

Table 5.2: Evaluation: Time measurements of the experiments. The table show the mean of five measurements.

Experiment number	required time in ms	standard deviation	computation result
1	542.4	1.1	1
2	3834.8	9.9	5
3	3834.8	19.8	5
4	1069.8	3.3	7
5a	539.8	2.0	3
5b	535.4	3.4	7
6	3783.4	44.0	5
7	584.6	5.6	7
8	543.2	1,5	3
9	572.8	25.76	5

Now we want to discuss the overhead that is created by NFN. Figure 5.13 visualizes the execution time of the different experiments. Thereby the execution time is split by using colors. The red parts of the execution time are the time, which a node waits for a timeout and cannot continue the computation. The blue parts are the time, which is required for the computation and the white parts are the time that is required for routing and transmitting CCN messages. The white parts are also shown in Figure 5.14.
The overhead of the timeouts occurs especially in heterogeneous networks with a lot of CCN-nodes to store the data. Additionally for short computations the timeouts are the longest part of the computation. This fact shows how important it is to create a NFN on top of an CCN with NACK support.

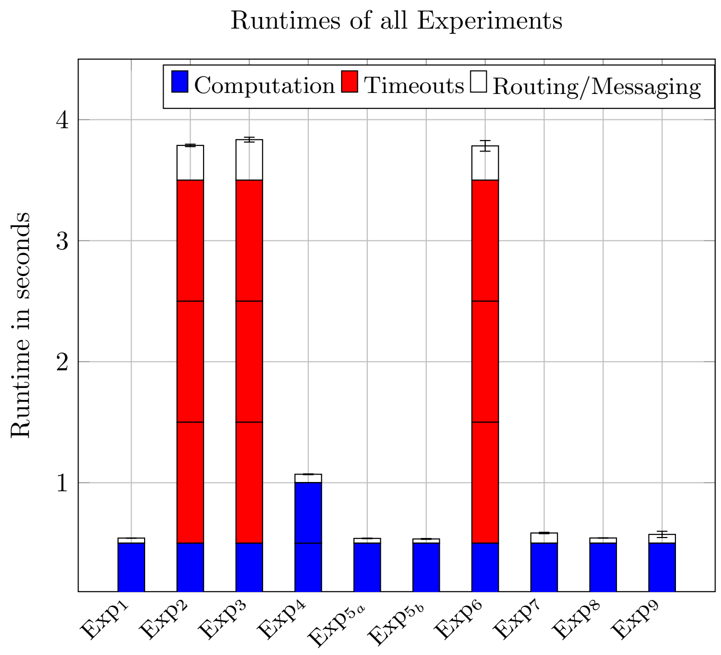


Figure 5.13: Runtime measurement of the experiments

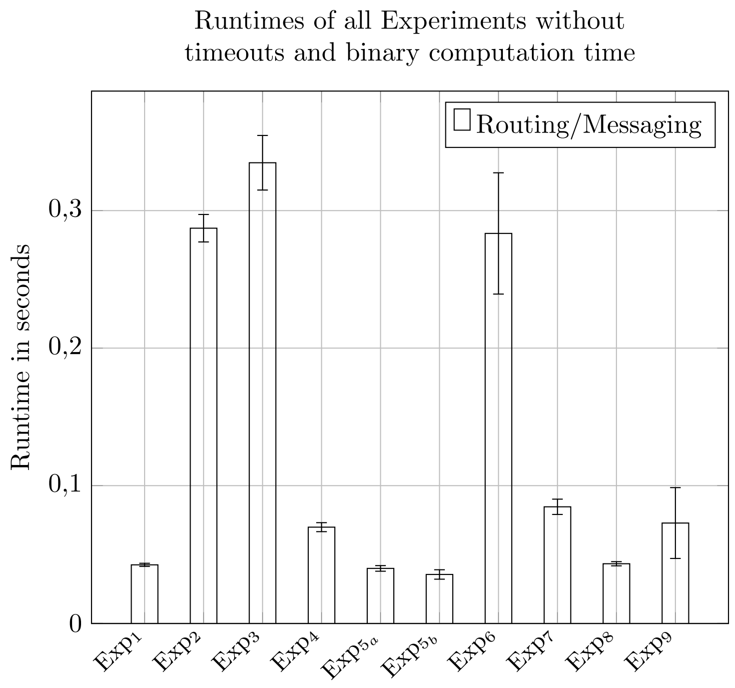


Figure 5.14: NFN overhead for routing and sending CCN messages

6
Discussion and Future Work

In this section we discuss the NFN implementation and design decisions. Additionally we give hints how the NFN system can be improved. We start with discussing the packet formats, next we talk about other ways to implement thunks or NACK messages. Furthermore we discuss the possibility to integrate more instruction into the NFN-engine to manipulate data with λ-expressions and how the implementation can be technically improved by extracting the NFN-engine out of the CCN-node and by multi threading the NFN-engine. At last there are some ideas are described, how publishing content and how to cryptographically sign computed results.

6.1 Packet Formats

Instead of creating new packet fields to mark an ICN packet that another packet type is included we integrate all information in the name. There were two main reasons for this. The first reason is that the CCNx protocol, on which we created our NFN implementation has no integrated support for additional marker fields. Hence, to keep the NFN-system compatible with the CCNx protocol it is required to use the existing packet fields. So it is possible to extend an existing CCN network with a few NFN-nodes to perform computations on the data. Since name based routing is based on longest prefix matching adding a NFN marker field at the beginning would destroy the routing system, thus we added the name components at the end, so that a CCN router ignores them. The name field of an interest message has an arbitrary number of components. Thus it is possible to encode an arbitrary computation in a name and to add additional fields to mark an interest as NFN-interest. Since there are other CCN implementations - e.g. NDN (Named Data Networking) - which also relay on name component based routing, it is possible to integrate the NFN implementation to other ICN implementations. An important requirement for this is, that the ICN system contains name based routing based on name components. Another possibility would be to integrate additional marker fields to each protocol, which should be used for NFN. But this would mean to design the fields for each protocol. So for portability it is more useful to integrate the marker fields and computations into the name components. An additional advantage of integration all required fields into the name components is that on this way it becomes easy

to create heterogeneous networks, which support different ICN protocols, if they are related, such as CCNx, CCN-TLV and NDN. A node can just write the name components and the content data into a specific packet.
To mark packets as NFN packet as well as to mark thunk packets we added an additional name component, but not for NACKs. For NFN-requests or thunks the user issues the interest and defines the naming component, in a way that the network can understand them. For NACKs this is not possible, because they are only a reply message. To route a NACK message back over the CCN network it is required to use the same name components. So it is possible to add a marker field for a NACK message or to write a specific string into the content object. There are ICN networks, which already support NACK messages, for example CCN-TLV. To keep the NFN protocol independent from the underlying ICN protocol we choose a specific string in the content object, which in our case is `:NACK`.

6.2 Thunk Implementation

Our NFN implementation strictly differentiates between result-request-packets and thunk-packets. If a user requests a result he will expect to receive a result in the reply message and if he requests a thunk he will expect to receive a thunk reply. This approach is very clean, it makes a first implementation easy and stable, but it has also disadvantages. If the final result is already available, but the user sent only a thunk request it would be possible to reply with the result. Instead, a thunk will be replied. Additionally, if a computation requires more time than the estimated value transmitted in the thunk, the thunk cannot be resolved and the computation will timeout. These problems can be addressed by unifying thunks and result-requests. If a user sends a request to the network he either receives the final result or a thunk which tells him that the result will be computed and how long it approximately would take. When the user requests the result again and the network is still computing it can transmit another thunk, so that the user can distinguish if the computation failed or requires more time. Since it is required to reply with exactly the name of the request, the CCN packet needs to be extended with a field to mark it is a thunk. It is possible to encode this field in the content object itself as we already did for the NACK messages (Section 4.3.3), so that the protocol stays compatible with CCN-only-nodes. It is recommend to avoid caching of thunks in this scenario. If a thunk message is cached it may be impossible to transmit the final result, because the thunk under the same name would be already available from cache. If for some reasons - e.g. because of the chosen CCN-protocol - it is impossible to avoid caching, one can use links to circumvent this problem. In this case a link contains the thunk timeout value and the name that should be requested to get the final result. If the result is still not available after the timeout the user will receive another link.
Unifying thunk and result-request messages could make the network more reliable in practice and gives the user more feedback over a running computation.

6.3 NACK Implementation

In Section 4.3.3 NACKs for the CCN as well as for the NFN were introduced. In our implementation we used only one type of NACK that tells the previous node that a resource cannot be delivered. NACKs speed up the reaction time of the network by avoiding timeouts. But there are further optimizations that could be applied to NACKs. Think about a network

that consist of a long chain of NFN-nodes. An interest was transmitted from the beginning of the chain to the end and the last node cannot satisfy the interest because a resource is missing and replies NACK message. If a resource is missing it is impossible to compute a result on another node, this would only be possible if there is a pinned function or the data are stored on an CCN-only-node. The NACK message is transmitted to the previous node which also cannot compute the result. This happens for all nodes in the chain. This example concerns all deep networks where long chains are possible. It would be enough if one NFN-node in the chain tries to compute the result. This is not necessary the last node in the chain because this node can be an ICN-node, but it can also be the first NFN-node in the list. This behavior can be achieved by using different types of NACKs, one if the sender was an CCN-node, and one if it was an NFN-node. If the last node is an CCN-node an CCN-NACK is returned. The first NFN-node that receives the CCN-NACK tries to compute the final result and replies with an NFN-NACK. Both CCN and NFN-nodes transmit a NFN-NACK like a normal content object and so only one node in the chain tries to compute the result. If the last node is already a NFN-node and the resource is not available it will reply with a NFN-NACK so that the other NFN-nodes know that they do not need touch the NACK message again.
Additionally, it is possible to encode why a NFN-NACK occur. If a resource is not available the computation can be directly aborted, but if it is just a pinned function the organizer can directly reroute the request to the node that stores the function. If an CCN-NACK occurs the organizer do not have to be involved since the NFN-node next to the CCN-node with the input data will compute the result.
Extending the functionality of NACK could further improve the reaction time of the network.

6.4 Data Manipulation in the NFN-Engine

The NFN-engine can only manipulate names and perform computations on numbers. The purpose of the NFN-engine is to distribute computations to different nodes. The computation itself is executed by the service layer. If a functionality for data manipulation is not available in the network the user cannot write it in the λ-calculus, but he has to add the function as a service layer function and publish it in the network. The main reason for avoiding data manipulation inside the λ-calculus is to avoid data in the name of a computation. But it would be possible to extend the NFN-engine in a way that it is possible to perform data manipulation in the λ-calculus inside a single NFN-engine. Thus no data are written inside an CCN name. A simple way for data manipulation would be to add a `head` and a `tail` function to iterate over a file. The `head` function returns the first character of a data object and the `tail` function the rest. To get the second character of a file the expression `head(tail(file))` can be used. A request could be `(λx0.head x0) /data/file`. First it is required to forward the request to a node which has the input file locally available. On this node the NFN-engine is invoked and can open the data file and compute the result. Thereby no network interaction occurs, so that no content is written in an ICN name, but the result can be returned within a content object to the requester. This way simple data manipulation could become possible by using the λ-calculus.

6.5 Packet Segmentation

To handle large data files in CCNx segments are used (2.1.1). Segments are part of the data file, which are encapsulated in separate CCNx packets. Similar concepts exist in other ICN protocols. To this point we did not talk about how to handle segments in NFN. The easiest way to handle segments would be using the service layer. If a data file consists of multiple segments the service layer requests all segments and applies the function on the segments. Thereby it is either necessary to combine all segments before the function is applied to get the correct result or to guarantee that all particular results of the segments are combined in the correct way. Computing the results on the segments separately and combining them later is similar to Map-Reduce operation. Thus this would be a way to introduce parallelism on the service layer. But is is not possible to define every program as Map-reduce operation. In this case it is required to fetch all segments and combine them to compute the correct result. Since the service layer communicates over the ICN protocol with the NFN-layer it is not possible to combine the segments already on the NFN-layer because then they cannot be transmitted to the service layer. For the NFN routing it is required to test if a content object is locally available. Therefore it is necessary to consider that it is possible that the input file was split into segments. This is important for matching the name with the input data, since segments usually have a further naming component, e.g. `<data_file_name>/segment1`. Furthermore if a data file is split into multiple segments the decision if the input data are available locally becomes more complicated, since it is possible that only a part of the input data file is available. In this case the network would have to check how many segments are not available locally, and if there is a node, which has more segments of the file available. The node, which has the most segments locally is the node that should start the computation, since than the number of transmitted segments is the smallest.

6.6 NFN-Engine Outside of the CCN-node

For our implementation we integrated the NFN-engine directly into the CCN-node. The advantage of this approach is that the NFN-engine can directly access the internal data structures of the CCN-node without communication. The disadvantage is if the NFN-engine crashes for some reason the entire node will also crash. This is one of the reasons why the service layer implementation is outside of the CCN-node. It would be also possible to extract the NFN-engine out of the CCN-node and place it behind a face. To do this it is necessary to define a communication protocol such as for the communication with the service layer. This protocol can use the CCN packets as basis. Further advantage of extracting the NFN-engine out of the CCN-node would be that it becomes possible to extend an CCN router with an NFN-engine by just installing a face to an external computation system, without changing the CCN hardware.

6.7 Handling Multiple Computations

To handle multiple computations at the same time the NFN-engine uses the PIT and a list of configuration. This procedure is described in Section 4.4.3. Inside the NFN-engine there is no real concurrency, but only one computation is active at the same time. Multiple computation are handled by evaluation them one after the other. If a computation cannot be continued, because a resource is missing the NFN-engine will pause this computation and continue with another one. Another way to handle multiple computations at the same time would be to use threads. The advantage of threads is that a computation does not have to wait until another computations stops. True concurrency will be introduced since the NFN-engine can start a new thread for every computation request. When an input data file for a computation arrives the CCN-node has to distribute it to the thread it belongs to. This can be done by the PIT entries. If a node sends a request to the network a PIT entry with thread id of the requesting thread is created and the thread goes in a waiting state. If the corresponding content object arrives the CCN-node will notify the thread to continue the computation. Using threads is beneficial if the system, on which the NFN-engine is executed has multiple processing units (e.g. CPUs or CPU cores). In this case the system would be able to execute multiple computations at the same time, not only to handle them.

6.8 Publishing Functions and Content in CCN/NFN

Publishing content in CCN is a well known problem. If one assumes every node will have a certain prefix a node can publish content under the prefix. If a user wants to request a file, he will has to prepend the prefix of the user. The problem is that to request a file the prefix of the publisher has to be known. Usually publishing functions is similar to publishing content. But especially with function, which are often stored on different nodes at the same time it would be useful to avoid using the prefix of the node. Thus it may be useful to introduce special prefixes and to install additional forwarding rules for functions. In practices special prefixes of function may be library names or name spaces, so that it is easy to find functions that can be used for a program. But what if a user wants to publish a new function? How

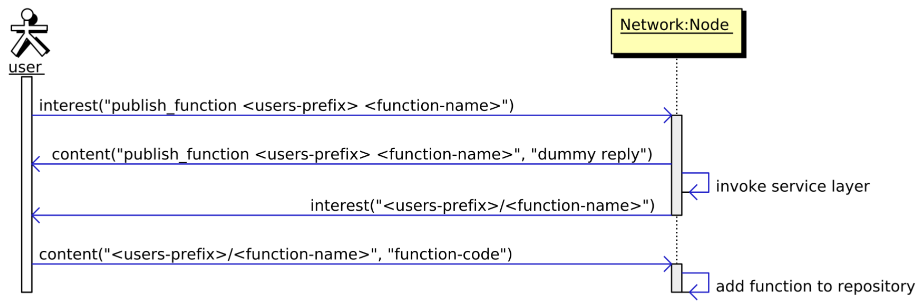

user
Network:Node
interest("publish_function <users-prefix> <function-name>")
content("publish_function <users-prefix> <function-name>", "dummy reply")
invoke service layer
interest("<users-prefix>/<function-name>")
content("<users-prefix>/<function-name>", "function-code")
add function to repository

Figure 6.1: Publishing functions on a library node in NFN.

is a new function available under the library name if there are only forwarding rules to the user for prefix of user's node? When a user publishes a function under a library name on a CCN-node other nodes may not find the function unless the routing system updates its forwarding rules. So it is required to assume that under the CCN system there is a routing algorithm, which creates the required entries in the PIT. Another way to publish a function is to call a pinned function *publish* on a node that already has the library prefix. As a consequence there will be library nodes in the network, which make functions available to other nodes. The publish function is called with the prefix of the user and requests the function to be published from the user by sending an interest message. Thereby the function will be published on a library node and not on the users node. Forwarding rules with the library prefix to the library nodes are already available in the network. Figure 6.1 shows this process. For all testing scenarios we installed additional forwarding rules for functions to store the them on different nodes.

6.9 Signatures for Computed Results

In CCNs cryptographic signatures are used to guarantee the authenticity of the delivered content. With the NFN system this is more complicated. How to sign the result of a computation? One way to do this is, that the content is signed by the node that performed the computation. A user receives a result of a computation which is signed by a specific node. Now it is up to the user to trust the node, or to ask for the result from another node (if the CCN protocol contains the functionality to exclude nodes). But for example a video trailer which was published from a big company should be converted to lower resolution. If the result is computed on a node of the company it can be signed by the company itself. But what if the result is computed on another node which does not belong to the company? As described above the content can be signed by the node that computed the result. But the content was not changed from the human point of view, only the resolution was changed. So would it be possible to sign the result with a signature of the original publishing company? Since the company must not share their signing key for security reasons, the result must be routed to a node of the company. But this can destroy the efficient routing system of the underlying CCN system. A middle way would be that the node that performed the computation signs the content with its signature and adds the signatures of the input content, so that the user which receives the result can track which input data where used. But it is still required that the user trusts the node that signed the content, since there is no guarantee that the node added the correct signatures and used the correct input data. It is required to research how the authenticity of computed results can be guaranteed.

7 Conclusion

In this work we showed how to extend a computer network to be programmable as a single machine. To achieve the goal we extended a CCN network to be able to satisfy requests for results of computations. CCN uses the publish/subscribe semantics to deliver content to the user. The network is not based on connections but it relies on names. A user gives a request (interest) for a named file in the network and the network finds out how to deliver the file (content object). To reduce the overhead created by FIB entries for every content object a hierarchical name structure is used. Hierarchical name base routing relies on longest prefix matching. It is not important which host delivers a file, but only a match of the names matters and so it is possible to deliver requested content from any node. Cryptograpfic signatures can guarantee the authenticity of the delivered content. This property enables the CCN to cache often requested content next to the user and thereby to reduce the load of the network and to improve the reaction time.
Requesting data from the network is only a special case of requesting the result of a computation. An expression written in the λ-calculus can be encoded in the name of a CCN interest message. The λ-calculus is a formal representation of computations based on a recursive structure of variables, abstractions and applications. It is Turing complete. Thereby the λ-calculus is handled by an abstract machine which is integrated into each CCN-node. We called an extended CCN-node a NFN-node and the abstract machine inside a CCN-node NFN-engine. The NFN-engine manipulates λ-expressions inside of a CCN interest name. An interest for a computation will be marked by a special flag so that a NFN-node can distinguish it from interests for content. To keep the routing capabilities of the CCN, it is not possible to manipulate the content of data objects within λ-expressions, but only numbers and names. The goal of the NFN-engine is to optimize the location where data manipulation takes place. To manipulate data objects an additional service layer system is introduced which runs on top of the NFN-engine. The service layer is a separate engine which executes high level functions, whereby data content objects from the CCN/NFN are used as input. A high level function can be written in an arbitrary programming language. This property enables users to integrate existing programs in the NFN without rewriting the entire program. The function, which are executed by the service layer are not fixed on a node and can be transferred over the network. This property enables the network to

compute results next to the data location.
Our NFN-system should to handle three different cases as described in[2]: First the NFN-system tries to deliver the result from the cache. If a result was already computed it can be delivered directly and no computation is required. Second if the result is not available in the cache of the network it has to be computed. Therefore, the request is routed to a node, which stores the input data locally. This reduces the network traffic because a result is usually smaller than the input data. Third if it is not possible to compute the result on the location where the data are stored, it will be required to transfer the input data to a node that can compute the result. These three steps are executed by the FOX (Find Or Execute) instruction. The FOX instruction is an additional instruction, which was included into the abstract machine to handle the network requests. To avoid unnecessary network traffic the node that performs the computation should be the node, which provides the input data. But the node with the input data cannot compute the result by itself if it is a CCN-only-node. In this case a NFN-node next to the node with the input data should compute the result. If the function is not available in the network because it is pinned to another node, the computation has to be forwarded to this node.
The first node that receives a compute request by the users becomes the organizer node of the computation. It is responsible that the computation will be finished. If it receives an interest message it will first check if there is already a routable name component prepended. A routable name is a normal CCN name that can be handled by every CCN-node. Since the input data for a computation are taken from the CCN, they are addressable by CCN names. So one of the input names can be prepended to the computation string and the longest prefix matching of the CCN-nodes will forward the interest to the data source. If there is no routable component prepended, the organizer node will choose a name to be prepended. If there are multiple input parameters a defined strategy will be used to choose the name to prepend. Every node that receives a computation interest, which already has a component prepended will not touch the interest and just forward it unless the prepended name is locally available. If the prepended name is locally available the node will start computing the result by fetching all other required resources, including the function. If the function is not available the result cannot be computed. The same problem occurs if the node, which stores the input data is a CCN-only-node. Such a node will not understand the computation request, even if it has the input data available. In this case the interest will timeout at the organizer node, and the organizer node chooses another parameter. On the other nodes, which forwarded the interest, the computation interest is removed after a timeout occurs, such as a normal CCN interest does. If it is not possible to compute the result by routing the computation to any input data, the computation will be routed to the function.
To reuse already computed results and intermediate results they have to be stored in the cache of the network. Since the network stores the results on the node, which computed it, searching for a cached result and forwarding the computation to a node, which stores the input data locally, is very similar and the first two steps of the FOX instruction become the same request. The node, on which the input data are stored either replies the result from cache or computes it. It is important only to cache steps that take longer to compute than the round trip time of a request or a timeout to benefit from the caching. Thus, caching and searching for every machine configuration is much to expensive. Our tests showed that caching and searching the results of a service layer computation is a good granularity to

benefit from the cache.
To avoid timeout for long running computations a notification system is introduced. A thunk message will verify if it is possible to deliver a result and how long it approximately takes. Thunk messages are marked by the NFN flag and an additional thunk flag. If a node receives a thunk reply it will know that the computation can be completed and can adjust its timeout interval by the value of the thunk. A computation can only be continued if all required thunks are available. To receive the final result a normal NFN-interest without the thunk request is used. The user has to trigger the normal NFN-interest as well as the thunk message. To compute subcomputations in parallel thunks can be used, too: If a node receives a thunk request it can already prepare the result while the organizer node requests other thunks (which also trigger a computation). Our tests with thunks showed that timeouts with long running computations can be handled by using thunks. Additionally, the tests showed a speedup for parallel computations.
To avoid timeouts at the organizer node and to compute the result as close to the node that stores the input data as possible it is required to notify the previous node, that a computation cannot be executed. This will be the case if the input data are stored on a CCN-node. An CCN-only-node cannot satisfy a computation request, thus it sends a NACK message to the previous node. If this node also is a CCN-only-node it will just forward the NACK message. But if the node is a NFN-node the node will understand the message as request to compute the result by itself. This way it was possible to avoid the timeouts at the organizer node and to avoid rerouteing the request to the next input data even if it would be possible to compute the result next to the input data, which were chosen first. Additionally, the reaction time of the network is reduced.
Our NFN system gave a user a simple tool to program networks similar to a local machine by using λ-expressions and a high level programming language. The network optimizes where a computation takes place. Depending on the current scenario the network can compute a result on different nodes. This property makes NFN an easy programmable and efficient system for computations in the network. Especially data processing is a strength of the NFN system, since for example Map Reduce operations can easily be executed in λ-calculus of NFN.

Bibliography

[1] Church, A. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):pp. 345–363 (1936). URL `http://www.jstor.org/stable/2371045`.

[2] Tschudin, C. and Sifalakis, M. Named Functions and Cached Computations (2013).

[3] Krivine, J.-L. A call-by-name lambda-calculus machine (2006).

[4] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 1–12. ACM, New York, NY, USA (2009). URL `http://doi.acm.org/10.1145/1658939.1658941`.

[5] Gritter, M. and Cheriton, D. R. An Architecture for Content Routing Support in the Internet. In *USITS*, volume 1, pages 4–4 (2001).

[6] Saroiu, S., Gummadi, K. P., Dunn, R. J., Gribble, S. D., and Levy, H. M. An Analysis of Internet Content Delivery Systems. *SIGOPS Oper. Syst. Rev.*, 36(SI):315–327 (2002). URL `http://doi.acm.org/10.1145/844128.844158`.

[7] Paul, S., Pan, J., and Jain, R. Architectures for the future networks and the next generation Internet: A survey. *Computer Communications*, 34(1):2 – 42 (2011). URL `http://www.sciencedirect.com/science/article/pii/S0140366410003580`.

[8] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M., Briggs, N., and Braynard, R. Networking Named Content. *Commun. ACM*, 55(1):117–124 (2012). URL `http://doi.acm.org/10.1145/2063176.2063204`.

[9] Center, P. A. R. Official CCNx project site (2013). URL `http://www.ccnx.org/`.

[10] Jacobson, V., Smetters, D. K., Briggs, N. H., Plass, M. F., Stewart, P., Thornton, J. D., and Braynard, R. L. VoCCN: Voice-over Content-centric Networks. In *Proceedings of the 2009 Workshop on Re-architecting the Internet*, ReArch '09, pages 1–6. ACM, New York, NY, USA (2009). URL `http://doi.acm.org/10.1145/1658978.1658980`.

[11] Perino, D. and Varvello, M. A Reality Check for Content Centric Networking. In *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking*, ICN '11, pages 44–49. ACM, New York, NY, USA (2011). URL `http://doi.acm.org/10.1145/2018584.2018596`.

[12] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10 (2010).

[13] Jung, A. A short introduction to the Lambda Calculus (2004).

[14] Cheriton, D. and Gritter, M. TRIAD: A New Next-Generation Internet Architecture (2000).

[15] Tschudin, C. and Sifalakis, M. Official CCN-lite project site (2013). URL `http://ccn-lite.net/`.

[16] Pouwelse, J., Garbacki, P., Epema, D., and Sips, H. The Bittorrent P2P File-Sharing System: Measurements and Analysis (2005).

[17] Turing, A. M. Computability and -Definability. *The Journal of Symbolic Logic*, 2(4):pp. 153–163 (1937). URL `http://www.jstor.org/stable/2268280`.

[18] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265 (1936).

[19] Diehl, S., Hartel, P., and Sestoft, P. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751 (2000).

[20] Landin, P. J. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320 (1964).

[21] Leroy, X. The ZINC experiment: an economical implementation of the ML language (1990).

[22] Friedman, D. P., Ghuloum, A., Siek, J. G., and Winebarger, O. L. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 20(3):271–293 (2007).

[23] Freed, N. and Borenstein, N. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies (1996).

[24] Berners-Lee, T., Fielding, R., and Frystyk, H. Hypertext transfer protocol–HTTP/1.0 (1996).

[25] Zhang, L., Estrin, D., Burke, J., Jacobson, V., K, J. T. D., Zhang, S. B., Dmitri, G. T. K. C., Massey, K. D., Papadopoulos, C., Lan, T. A., Crowley, W. P., Zhang, L., Estrin, D., Burke, J., Jacobson, V., Thornton, J. D., Smetters, D. K., Zhang, B., Tsudik, G., Claffy, K., Krioukov, D., Massey, D., Papadopoulos, C., Abdelzaher, T., Wang, L., Crowley, P., and Yeh, E. Named Data Networking (NDN) Project NDN-0001 (2010).

[26] Vinoski, S. Distributed object computing with CORBA. *C++ Report*, 5(6):32–38 (1993).

[27] The Message Passing Interface (MPI) standard (2014). URL `http://www.mcs.anl.gov/research/projects/mpi/`.

[28] Hadoop, A. Hadoop (2014). URL `http://hadoop.apache.org`.

[29] Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D., and Minden, G. A survey of active network research. *Communications Magazine, IEEE*, 35(1):80–86 (1997).

[30] Smith, J. M., Farber, D. J., Gunter, C. A., Nettles, S. M., Feldmeier, D., and Sincoskie, W. D. Switchware: Accelerating network evolution (white paper) (1996).

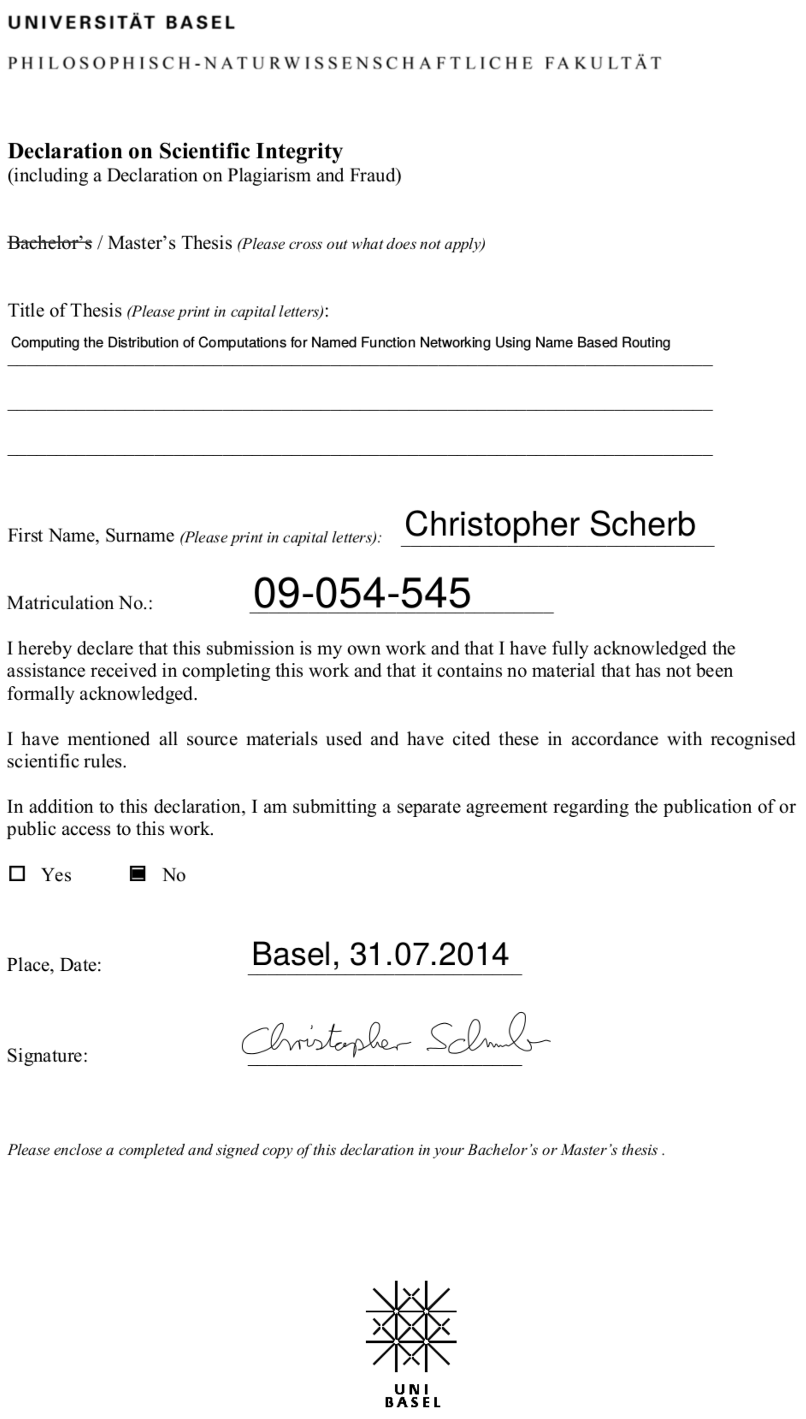
[31] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74 (2008). URL `http://doi.acm.org/10.1145/1355734.1355746`.

[32] Tschudin, C. Fraglets-a metabolistic execution model for communication protocols. In *Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA*, volume 6, pages 1–3. Citeseer (2003).

[33] Microsoft. Map/Reduce - in Functional Programming Parallel and Processing Perspectives (2009). URL `http://blogs.msdn.com/b/csliu/archive/2009/11/10/map-reduce-in-functional-programming-parallel-processing-perspectives.aspx`.

[34] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113 (2008). URL `http://doi.acm.org/10.1145/1327452.1327492`.

[35] Kohler, B. *Data Computation in Named Function Networking based on High-Level Programming Languages*. Master's thesis, University Basel, Switzerland (2014).



UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)

~~Bachelor's~~ / Master's Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:
Computing the Distribution of Computations for Named Function Networking Using Name Based Routing

First Name, Surname *(Please print in capital letters)*: Christopher Scherb

Matriculation No.: 09-054-545

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

☐ Yes [✓ No]

Place, Date: Basel, 31.07.2014

Signature: [illegible]

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .

UNI BASEL