Equivalence Classes for Named Function Networking

Inauguraldissertation

zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Urs Schnurrenberger
aus Sternenberg ZH

2019

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Christian F. Tschudin, Universität Basel, Fakultätsverantwortlicher
Prof. Dr. Jussi Kangasharju, Universität Helsinki, Korreferent

Basel, den 23.04.2019

Prof. Dr. Martin Spiess, Dekan

Abstract

Named Function Networking (NFN) is a generalization of Content-Centric Networking (CCN) and Named Data Networking (NDN). Beyond mere content retrieval, NFN enables to ask for results of computations. Names are not just content identifiers but λ-expressions that allow an arbitrary composition of function calls and data accesses. λ-expressions are pure and deterministic. In other words, they do not have side effects and they always yield the same result. Both properties together are known to as *referential transparency*. Referentially transparent functions can be evaluated individually no matter where and in what order, e.g. geographically distributed and concurrently. This simplifies the distribution of computations in a network, an attractive feature in times of rising needs for edge computing. However, NFN is affected by a lacking awareness for *referentially opaque* expressions that are characterized by having changing results or side effects, i.e. expressions that depend on outer conditions or modify outer states.

The fundamental motivation of this thesis is to retrofit NFN with a clearer notion of referentially opaque expressions. They are indispensable not only to many common use cases such as e-mail and database applications, but also to network technologies such as software defined networking. We observed that many protocol decisions are based on expression matching, i.e. the search for equivalent expressions. Driven by this observation, this thesis explores possibilities to adapt the determination of equivalences in dependence of crucial expression properties such as their ability for aggregation, concurrent evaluation or permanently cacheable results. This exploration results in a comprehensive set of equivalence classes that is used for explicit attribution of expressions, leading to a system that is aware of the true nature of handled expressions. Moreover, we deliver a solution to support referentially opaque expressions and mutable states in an architecture that bases upon uniquely named and immutable data packets.

Altogether, the findings condense to an extended execution model. It summarizes how the attribution of expressions with equivalence classes influences specific protocol decisions in order to support referentially transparent as well as referentially opaque expressions. We believe that our approach captivates due to its generality and extensibility. Equivalence classes depend upon universal properties. Therefore, our approach is not bound to a specific elaboration like NFN. We evaluate the applicability of our approach in a few application scenarios. Overall, the proposed solutions and concepts are an important contribution towards name-based distributed computations in information-centric networks.

Acknowledgments

Special thanks go to Christian Tschudin for granting great flexibility to organize my work. A maximum of academic freedom has always been ensured. His encouragement for self-reliant working as well as his scientific opinion were inspiring and much appreciated. Not least because of that I will keep this time in good memory.

Many thanks go to Jussi Kangasharju for his independent assessment of this work and for contributing another valuable scientific opinion. The great deal of work it involves is not taken for granted and much appreciated, too.

Table of Contents

List of Figures

List of Code Listings

List of Tables

List of Acronyms

AI	Arrival Interface
AR	Augmented Reality
BGP	Border Gateway Protocol
CAS	Computer Algebra System
CCN	Content Centric Networking
CDN	Content Delivery Network
CET	Central European Time
CmRDT	Commutative Replicated Data Type (a.k.a. operation-based)
CRDT	Conflict Free Replicated Data Type
CS	Content Store
CvRDT	Convergent Replicated Data Type (a.k.a. state-based)
(D)DoS	(Distributed) Denial-of-Service
DNS	Domain Name System
DONA	Data-Oriented Network Architecture
EC2	Elastic Compute Cloud
ECS	Equivalence Class System
FIB	Forwarding Information Base
FOX	find-or-execute
FTP	File Transport Protocol
HIP	Host Identity Protocol
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IaaS	Infrastructure as a Service
ICN	Information-Centric Networking
IGMP	Internet Group Management Protocol
IMAP	Internet Message Access Protocol
IoT	Internet of Things
IP	Internet Protocol
KS	Kernel Store
LPM	Longest Prefix Match
MEC	Mobile Edge Computing, Multi-access Edge Computing
MSI/MESI/MOESI	Modified-((Owned-)Exclusive-)Shared-Invalid protocol
NDN	Named Data Networking
NFaaS	Named Function as a Service
NFN	Named Function Networking
OOP	Object Oriented Programming
PaaS	Platform as a Service
PIT	Pending Interest Table
POP	Post Office Protocol
QoE	Quality of Experience
REST	Representational State Transfer
RICE	Remote Method Invocation in ICN
RO	Referential Opacity / referentially opaque
RT	Referential Transparency / referentially transparent

RTT	Round-Trip Time
S3	Simple Storage Service
SaaS	Software as a Service
SCN	Service Centric Networking
SML	Simple Mini Language
TLS	Transport Layer Security
TLV	Type-Length-Value
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
VM	Virtual Machine
WWW	World Wide Web
XaaS (a.k.a. EaaS)	Everything as a Service

1. Introduction

This thesis develops the idea of name-based computations in information-centric networks (ICN). ICN is characterized by data with unique names. This so-called named data can be fetched by looking up its name in the network. Beyond that, Named Function Networking (NFN) is an approach that combines requests for named data and the evaluation of expressions in order to actively perform computations, not only lookup. Subexpressions are evaluated and replaced by the names of their results. This procedure is continued until all subexpressions are resolved and the final result is available. However, this approach only works if subexpressions are referentially transparent. Referential transparency means that an expression can be replaced by its value without changing the outcome of a computation, i.e. without changing its result. If this is not the case, the expression is said to be referentially opaque.

Unfortunately, NFN and other current approaches do not differentiate between referentially transparent and referentially opaque expressions. In the better case, this only leads to inefficiencies. In the worse case, it leads to wrong results. Instead of limiting ourselves to referentially transparent expressions, we argue that we should actively support referential opacity. To do so, we promote to always perceive names as expressions and to attribute them with equivalence classes. This enables the network to be aware of a request's true nature and to act accordingly.

1.1 Problem Statement & Motivation

In today's highly computerized world, more and more services are digitalized and shifted to the Internet. An avalanche of devices is connected to the Internet every day, causing whole new networking demands, particularly in terms of bandwidth and processing power. Moreover, new use cases and scenarios arise from spheres like the Internet of Things (IoT). Examples are connected cars and smart home automation, accentuating demands such as low-latency and high-availability. Cloud computing as an established technique struggles especially with data-intensive and low-latency scenarios, mainly because cloud computing infrastructure is far from where data is used but usually also far from where it is produced. Although the cloud does not have problems to provide enough computing power, it has problems to provide it in the right place, i.e. where it is needed. Computations should take place where data accrue. For example, data from connected cars should be processed as close to the cars as possible, involving a minimum of near-by communication infrastructure due to latency. Hence, it reveals that also computations must be more flexible, i.e. mobile and not only reserved for cloud service providers.

New Paradigms

Several approaches try to cope with these new demands on the network level. Prominent examples from the *name-based content retrieval* universe are Content-Centric Networking (CCN, (1)) and Named Data Networking (NDN, (2)). In contrast to the predominant Internet Protocol (IP), their specialty is to request information by name instead of by network location. Newly introduced

stateful in-network data-structures allow omnipresent data caching, and therefore close to where it is used, as well to retrieve content from multiple locations at the same time. With those features, especially bandwidth, latencies and availability concerns are addressed.

From Content Towards Computations

Unfortunately, alternatives to the IP traditionally focus on the transport of data only. This also applies to CCN/NDN, missing the computing aspects entirely. It is often mentioned that the IP was designed for a different time with different user demands. Consequently, a new networking paradigm was presented that covers the problems of the past and the present. However, what is missing is the readiness for the future, i.e. the support for in-network computations. Foreseeing the future, Named Function Networking (NFN, (3)) addresses this need. It is a generalization of name-based content retrieval systems. Static content is just a special case of a result. Names express rather computations than mere content identifiers. In NFN, computations can be shifted through and spread over the network while it is up to the system to decide whether to pull data towards computations or to push computations towards data. Therefore, it is predestined to close the gaps between data producers, data processors and data consumers.

Inherent Support for Mutable States

However, NFN is affected by a lacking awareness for expressions that alter or whose results are altered by *mutable states*. This causes problems because everyday services and applications like HTTP, e-mailing, and databases in general, likewise produce and work with mutable states. Despite this, NFN as well as CCN/NDN are specialized in immutable states. Requesting twice the same expression or name must resolve to the very same result. Instead of implementing arbitrarily complex and service specific solutions on top of the architecture, we argue the architecture itself should provide the necessary functionalities built-in to support such everyday services. Consequently, we ultimately aim at a system that is aware of the true nature of handled expressions, facilitating a more expressive and efficient system.

1.2 Design Rationale

Rethinking Protocol Design Principles

The design principles of information-centric networking approaches have been chosen with content retrieval only in mind, e.g. such as for NDN (4). These design principles are set in stone, i.e. whether changed nor extended ever since. Although this practice guarantees a minimum degree of consistency and continuity, it leads to a potpourri of special-purpose solutions whenever new ideas or functionalities shall be supported. We have a different mindset and argue that, if there is a well-founded reason, it is more meaningful to adapt design principles to new demands instead of starting over to create a rag rug of special-purpose solutions. And we argue that the idea to shift name-based computations at the network layer into and at the edge of networks is such a well-founded reason.

Generality of the Approach

Although our approach is motivated to enhance the concept of NFN, i.e. the intermixing of λ-calculus for expression resolution and binary function calls for efficient computations, our concept shall not be bound to a specific approach and applicable to other approaches as well. Likewise, we focus on required adaptions to the architectures of CCN and NDN because they are the dominant elaborations among name-based network layer surrogates. However,

this shall not implicate to be bound to these specific systems. Therefore, we decouple theory from architecture and application and address the topics individually.

Explicit Attribution & Concepts

We pursue the approach to explicitly equip expressions with attributes that reflect certain concepts. For example, imagine the problem of concurrent function evaluation. There are three options how to face up with it: 1. Leave the system non-concurrent, i.e. sequentially. 2. Restrict the system to only concurrent operations, e.g. such as purely functional approaches do, or 3. differentiate between functions that can be evaluated concurrently and those that cannot, e.g. by tagging them. For example, Fortran's *pure* keyword is an exponent of the third option. The keyword declares a function to be free of side effects and simultaneously signifies the compiler that it can flexibly order the evaluation of such functions. Therefore, it can optimize towards parallel computations. Also note the generality of the keyword. Although it is a specialty of Fortran, the concept behind the keyword is not bound to the language and can be transferred to other languages as well. For clarification, note the difference between a fictive 'can-be-parallelized' keyword and 'pure'. While the former describes the consequence, the latter describes the reason.

We try to achieve the same for our attributes, i.e. to expose certain expression qualities, independent of a specific architecture. In this way, every system can decide by itself what the concrete consequences are, e.g. if a result is permanently cacheable or not. Another advantage of explicit attribution is that the requester is enabled to explicate, and implicitly also to exclude, specific effects of its request. Therefore, it cannot be held accountable for unforeseen and potentially harmful additional effects of its request, e.g. due to a faulty implementation. For instance, this idea is also followed by the definition of *safe* methods in HTTP, the protocol's approach to summarize methods that should not have side effects (5).

Terminology

Throughout this thesis, certain terms will appear frequently. Some of them have identical meanings and are therefore used interchangeably. Depending on the context, we use the most appropriate term.

- **Interests**, short for interest packet, are also called *requests* and **data packets** are also referred to as *responses*, *replies*, *result packets* or *content objects*. Result packets and content objects do not differ in their appearance. However, the former implies a prior computation.
- Both interest and data packet contain a **content name**, also referred to as *name*. The content name is always an **expression**, even if the expression is nothing but a (content) **identifier**. However, the expression can also contain further (sub)expressions. For example, a function call can contain argument expressions. Hence, an expression is not always a content name.
- Among other packet fields, data packets carry **data**. Data is also called *content*, *result*, or *payload*. A result is again the outcome of a prior computation. Other packet fields like *signature* or *nonce* are named unambiguously.
- There is always a lot of debate about the following terms. This is how we use it: A **routine** is either a **function** that is defined by having a return value but no side effects or an **action** that is defined by not having a return value but side effects. If the routine has unknown consequences or both a return value and side effects, we keep calling it routine. Although we do not make

use of the following terms, they shall be mentioned for the sake of completeness: A *method* is the same as a routine if associated to an object and actions are also called *procedures*. Hence, there are *function methods* and *procedure methods*.

- Expressions are commonly said to be **evaluated**. However, expressions can also be perceived as small programs, especially if several expressions compose larger expressions, which are rather *executed* than evaluated. Related terms are **calling** and *invoking* routines or programs, causing their evaluation or execution.
- Basically, results cannot be **transparent** or **opaque**. However, if we speak from transparent or opaque results, we mean results being produced through the evaluation of referentially transparent or referentially opaque expressions. In general, transparent results are *persistent* and opaque results are *transient*.

1.3 Contributions of the Thesis

The goal of this thesis is to provide a general theoretical solution to perform distributed name-based edge and in-network computations. The main contribution of this thesis is the extension of this scheme towards a system that is aware of both referentially transparent and referentially opaque expressions. Especially being aware of referential opacity is essential for a system that should act according to expectations. Our equivalence classes describe a universally valid set of expression properties. Making use of equivalence classes reduce the problem of differentiating and supporting all kind of expressions to a matching problem, i.e. the context-dependent determination of equivalences between attributed expressions. The specific contributions are:

- Although content names in CCN and NDN are unique and therefore also unambiguous, we show that the according content retrieval mechanism allows ambiguous requests. Hence, responses in CCN and NDN are not necessarily deterministic.
- We show that NFN is not immune to non-deterministic results, too. Beyond determinism, NFN also struggles with side effects. Both originate particularly from NFN's lacking awareness for referentially opaque expressions.
- We introduce a new and extensible classification system for expressions. So-called equivalence classes are derived from a set of expression properties. For each equivalence class, a different subset of properties applies.
  - We explain how to use *syntactic* equivalence correctly in order to reach truly deterministic content retrieval.
  - We give advice how to make use of *sematic* equivalences between expressions and provide insights into chances and limitations.
  - We cover the space of referentially opaque expressions with one *adaptive* equivalence class. This means that equivalences are adaptively determined based on expression subclasses and current network tasks. Expression subclasses emerge from deconstructing results into return value and side effect and an individual examination of the two constituents.

  - For a first subclass arising from this separation, we show how to handle expression with only *ephemeral* return values.
  - We further differentiate side effecting expressions into unproblematic and problematic categories, highlighting the benefits of *commutativity* and *sequence-idempotence* for distributed name-based computations.
  - Finally, we show how to work with expressions that are *problematic* in terms of distributed evaluations. These are expressions that manipulate shared mutable states in a way that only sequential evaluation is applicable.
- We attached great importance to create a flexible attribution design. Therefore, the same way how we attribute equivalence classes to expressions can be used to attribute further individual and expression-specific attributes.
- We answer the question how different equivalence classes influence each other, particularly for cases where they appear in the same expression, i.e. nested and sequentially composed expressions.
- We point out how equivalence classes facilitate to build a system that supports named functions, no matter if transparent or opaque. Mainly, this comprises required protocol adaptions. However, also architectural components of CCN/NDN are concerned. We provide an extended execution model, illustrating protocol adaptions in condensed form.
- To evaluate our approach, we invented a simple programming language that integrates the concepts of equivalence classes. Based on our language, we demonstrate how concrete applications can profit from the proposed attribution mechanism.

1.4 Thesis Outline

The thesis is organized into the following core chapters. To get an idea of what to expect in each chapter, individual summaries of the main contents are provided, too.

- **Chapter 2: Background**
  The chapter provides a historical roll-up from established networking technologies and explains why they do not match anymore today's demands. This leads to recent technologies like CCN, NDN and NFN that address these demands. As this thesis aims to leverage the edge/fog computing capabilities of NFN, the chapter necessarily covers the topics cloud, edge and fog computing as well as the IoT. Moreover, one of our key concerns is the lacking support for contents or results that change and functions that alter state outside of their scope. Therefore, the chapter contains a final section to explain side effects, deterministic results, and referential opacity in general.

- **Chapter 3: The Soft Spots**
  This chapter identifies where NFN has conceptual problems when being used in conjunction with referentially opaque functions. It also highlights that the content retrieval mechanism of CCN and NDN is not necessarily deterministic. Limitations of computability are being discussed, too. This is important when dealing with hardware representations of floating-point

numbers and mathematically difficult expressions that can only be approximated iteratively.

- **Chapter 4: Equivalence Classes**
  Our approach to make name-based computations aware of referential opacity and to support the resolving system in task-specific decisions, we enhance expressions with attributes that signify different expression qualities. The attributes implicitly tell the nodes how to interpret expressions, e.g. which of them are parallelizable and cacheable, as well as how to match them context-dependent against each other. This chapter covers theoretical aspects of the equivalence classes and finally summarizes them in the Equivalence Class System (ECS).

- **Chapter 5: Protocol & Architecture Adaptions**
  This chapter examines the consequences of the theory to the praxis. A concrete notation for the attribution is given first. Moreover, it is discussed how composed expressions must be treated, i.e. how subexpressions affect the equivalence class of the overall expression. Consequences to forwarding are explained consistently for all equivalence classes, while interest aggregation, caching and content matching is examined individually. The findings are summarized in an *extended execution model.*

- **Chapter 6: Evaluation**
  At the beginning of this chapter, a simple programming language is presented that enables ECS tagged network level requests for contents and computations. Moreover, the attribution helps to distribute computations over the network and to evaluate expressions concurrently. Three subsequent examples make use of the language, showing how the equivalence classes work in practice and what they can achieve, with a focus on scenarios using referentially opaque expressions.

- **Chapter 7: Related Work**
  There are only a few related works that are performing computations based on content-centric / named data networks. This leaves room for a thorough inspection of three approaches. Apart from caching, it is in focus how they handle referential opacity and how they make use of names to support computation-related decisions. As the topic of referential opacity received very little attention in these works, a final section is dedicated to well-known content retrieval and service invocation approaches that take the opposite perspective of NFN, namely the general assumption of transient contents and results.

- **Chapter 8: Discussion & Outlook**
  The thesis is closed with a discussion on open issues, ideas how this work can be developed further, and a conclusion.

2. Background

The background chapter provides insightful information about technologies and prior research that are fundamental to understand subsequently presented contributions. Advantages and shortcomings are discussed likewise, further motivating our proposed approach.

The chapter starts with a historical roll-up, leading from host-oriented and sender driven networking protocols like the Internet Protocol (IP) to more data-oriented approaches like publish/subscribe and Data-Oriented Network Architecture (DONA). Fully information-centric and receiver driven solutions like Content-Centric Networking (CCN) and Named Data Networking (NDN) are the essence of the second section. The third section highlights the generalized point of view taken by Named Function Networking (NFN). Implementation details provided later help to understand conceptual problems. Moreover, NFN is a very natural approach to edge and fog computing, necessitating a fourth section about these concepts, as well as their relation to cloud computing, the dominant way to outsource computations up to the present. Referential opacity (RO) is a topic that must be carefully considered when it comes to computations, especially in distributed settings. The fifth section introduces the theoretical background to it. All this background knowledge is used in chapter 3 to point out several spots in NFN and NDN that need improvement.

	Chapter	Section
	Background	2.1 From Host- to Information-Centricity
	The Soft Spots	2.2 Information-Centric Networking
	Equivalence Classes	2.3 Named Function Networking
	Protocol & Architecture Adaptions	2.4 Cloud & Post Cloud Computing
	Evaluation	2.5 The Risks of Referential Opacity
	Related Work	

TL;DR – Key Messages

✓ CCN and NDN request contents by names instead of over host addresses as the IP. They are receiver driven and stateful, enabling in-network caching and multipath routing. Moreover, they have strong security properties because content itself is secured instead of the connection.

- ✓ NFN is a generalization of CCN/NDN that enables to ask for results of computations. Static content is just one type of result. Names are λ-expressions rather than content identifiers. They can be rewritten to express resolution preferences.
- ✓ The cloud has strong centralization tendencies and advertises an outsource-everything-to-the-cloud strategy: data, services and even infrastructure. Unfortunately, this creates huge bandwidth demands and does not meet demands for low latencies, two aspects that gain(ed) importance with the rising of the IoT. Edge and fog computing promise remedy. They perform computations where it suits best to close the gaps between data producer, data processor and data consumer.
- ✓ referentially transparent ⇔ ¬side effects ∧ (¬non-)deterministic
  referentially opaque ⇔ side effects ∨ non-deterministic
- ✓ Inspired by CRDTs, commutativity, associativity and idempotence provide attractive features to perform distributed computations.

2.1 From Host- to Information-Centricity

Table 2.1 can be consulted as a small guideline for this section and the evolution from host- to information-centric protocols.

Table 2.1 – From IP to CCN/NDN

The table shows the development from the host-oriented and server driven Internet Protocol towards the data-oriented and receiver driven Content-Centric Networking / Named Data Networking by means of selected features.

	Host-oriented	Data-oriented	On-demand retrieval	On-path caches	Sender driven	Receiver driven
IP	✓	✗	✓	✗	✓	✗
Pub/Sub	✓	✓	✗	✗	✓	✗
DONA	✓	✓	✓	✓	✓	✗
CCN/NDN	✗	✓	✓	✓	✗	✓

Network Layer

Today's Internet is often described as model of abstract layers, covering tasks from application down to transferring data between neighboring nodes. Some models, such as the Open Systems Interconnection (OSI) model (6) or the Five-layer Internet protocol stack described by Kurose and Ross (7), include an additional hardware or physical layer while others, e.g. the Internet Engineering Task Force (IETF) Internet Standard document about Communication Layers (8), do not. However, all mentioned models include an Internet or Network layer. This layer summarizes protocols which organize the transport of network packets from one host to another host across network boundaries. Broadly speaking, a protocol on the Internet layer must distinguish different hosts from each other and it must provide a mechanism that guides packets between the two involved endpoints. The first requirement is called

addressing while the second requirement is called path selection or *routing*. In total, these two functions are the core of what is generally considered as the Internet.

Internet Protocol

The predominate protocol on the Internet layer is the Internet Protocol (IP). It differs hosts through IP addresses. The IP only defines the format of the addresses. The protocol itself is not responsible for assigning addresses to hosts. This can be done manually or through other protocols like the Dynamic Host Control Protocol (DHCP) (9). In case of IP version 4 (IPv4), the address is a 32 bit value (10), in case of IP version 6 (IPv6), it is a 128 bit value (11). Likewise, the IP is not responsible for gathering routing information. However, it is responsible for making routing decisions based on the destination IP address and node-local routing information.

Host Centricity

The idea of assigning addresses to hosts and organizing communication on top of them must be seen in the light of the era the protocol was developed. Before packet switching networks appeared in the late '60, circuit switching telephone networks have been the only possibility for bidirectional real-time communication. Their main purpose was to connect exactly two host devices (telephones). This host centricity remained valid for the first packet switched networks like the ARPANET, mainly developed to shift bulk data between two computers. Given these circumstances, assigning addresses (telephone numbers, IP addresses) to devices was the right choice.

Addressing Methods

However, it soon became clear that *unicast* packet delivery, i.e. transferring packets from the source to exactly one destination, is too strict for many applications. For instance, chat applications or sending control messages to a group of devices (hosts) could profit from more flexible packet delivery services. Reacting to such new demands, the Internet Protocol and associated protocols underwent several changes and extensions. This lead to the definitions of additional routing schemes like *broadcast* (12), *anycast* (13), *multicast* (14) and the newer *geocast* (15). Broadcast sends copies of a request message to all hosts within a local network. All receiving nodes reply to the message. Anycast and multicast both send copies of the request message to a (predefined) group of hosts. In anycast, only the host with shortest path to the source replies, while in multicast, all hosts reply. All these addressing methods are part of the Internet layer as well. Despite all innovations (or for just that reason), the protocol's host centricity survived.

Drawbacks

However, physical resources are no longer what users care about. The resource users care about nowadays is information. Browsing through the World Wide Web (WWW) is one of the most popular applications of the Internet and the best proof for the above statement. Usually, browsing the web happens by typing in or clicking on Uniform Resource Locators (URLs). However, it generally goes unnoticed that a certain part of the URL gets translated to an IP address by the Domain Name System (DNS). Even if noticed, the IP address will most likely not be double-checked before sending the request, simply because no one cares which server delivers the reply, as long as it is a legitimate server. Furthermore, there are technical reasons arguing against host addressing. For example, anycast is based on a deception of the Border Gateway Protocol (BGP). It uses the same anycast IP address for multiple hosts that are then interpreted by BGP as different routes to the same machine. BGP choses, according to some metric, the best path to the alleged singular host. For multicast to work, hosts must register themselves to a

multicast group. Therefore, another protocol is needed, e.g. the Internet Group Management Protocol (IGMP). A further reason is the insufficient support of host mobility. Whenever a host reattaches to the network in a different location, a new address will be assigned to it. This causes reachability problems, especially in session-based communications. The statelessness of the IP can also be discussed controversially. It prevents the aggregation of requests and replies on the network level, leading to a waste of transmission resources and ultimately to a reduced goodput[1] for every host. Therefore, the overall Internet experience is deteriorated. As a last reason, IP does not support caching on the network level. The protocol is, apart from source and destination addresses, agnostic of other packet contents and has therefore no possibilities to make caching decisions. This is another missed opportunity to improve throughput and latency.

Publish/ Subscribe

Not surprisingly, new ideas for the network layer emerged, some of them with 'data' instead of 'hosts' as their central building block. An early attempt to advocate a data-oriented networking perspective was the publish/subscribe pattern. Its core idea is that data consumers, i.e. *subscribers*, subscribe themselves to information classes or categories, while data producers, i.e. *publishers*, categorize data into information classes during the process of publishing. Many publish/subscribe implementations rely on message-oriented middleware, or more specifically on *message brokers* that coordinate subscriptions and message delivery. The pattern provides intrinsic support for multipoint-to-multipoint message delivery. Therefore, it principally finds application in group messaging, including group chats, mailing lists, news aggregators and blogs. The Rich Site Summary (RSS, a.k.a. Really Simple Syndication) is a well-known subscription system that implements the publish/subscribe pattern. However, the pattern is not a replacement of the Internet layer but runs above it. Furthermore, the pattern exhibits a weakness when it comes to on-demand information retrieval like browsing the web. Only publishers chose when data is sent. Subscribers only have the choice to filter categories, i.e. to receive all or nothing.

Information Centricity

These two shortcomings, still relying on IP and missing support for on-demand data/content/information retrieval, motivated further approaches. An early work trying to overcome both shortcomings, was the Data-Oriented Network Architecture (DONA) from Koponen et al. (16). They asserted that the Internet is rather used for "data retrieval and service access" than for "host-to-host applications such as telnet and ftp". Hence, they combined several ideas from prior works to build an approach more suitable for these new demands. For example, they resumed an idea from Cheriton and Gritter's TRIAD paper (17) to *route* requests *by name* to the closest copy. This contrasts with DNS' *lookup-by-name*, returning "the location (IP address) of a nearby copy" (16). Names are resolved directly instead of first mapping them to IP addresses. In fact, DONA's route-by-name resembles IP anycast directly on names above the IP layer, replacing DNS name resolution. Koponen et al. (16) also showed how this primitive can be used to reduce the complexity of supporting host mobility and multihoming. Data can be sent back to the requester *off-path*, i.e. on a different way than the request, using normal IP routing. However, DONA

[1] Understood as the application layer throughput, i.e. the network throughput in bits/s minus protocol overheads (IP, TCP, NDN, …) minus retransmissions. Note that there are other definitions like (109) that only subtract lost or retransmitted bits.

also supports (quasi) *on-path* routing. While routing a request towards the data, Autonomous System (AS) labels are collected. This path information can then be included in the reply. In other words, on-path routing means that data routing is *coupled* to name resolution while off-path routing means that data routing is *decoupled* from name resolution.

Another interesting idea is to extend all resolution handlers, i.e. routers, with a cache module. This universal general-purpose on-path caching infrastructure should enable new ways of improved content delivery.

2.2 Information-Centric Networking

Following DONA from 2007, several projects basing on 'named data' or 'named content' emerged, e.g. 4WARD (18) in 2008, Content-Centric Networking (CCN, (1)) in 2009 or Content Mediator Architecture for Content-Aware Networks (COMET, (19)) in 2010. Information-Centric Networking (ICN) is an umbrella term for those and other projects. A survey paper from Xylomenos et al. (20) discusses commonalities and differences of several ICN approaches by means of the following five categories: 1) naming, 2) name resolution and data routing, 3) caching, 4) mobility, 5) security. We forgo a detailed comparison of all different approaches as done by Xylomenos. However, we reuse the categories to describe the main characteristics of the related Content-Centric Networking (CCN) and Named Data Networking (NDN, (2)). Xylomenos et al. (20) call CCN the predecessor of NDN. We agree with this statement because NDN is made up of the same internal components and data structures as CCN. However, NDN should not be called the successor of CCN. More suitable is to call it a fork because the development of CCN proceeded after NDN has been lunched. The fork had rather theoretical reasons than a disagreement on architectural components. This thesis rather focuses on NDN than CCN. However, many insights will directly apply to CCN too. Whenever there is a significant difference between the two approaches, we will mention it explicitly.

2.2.1 Content-Centric Networking & Named Data Networking

Two years after DONA, Jacobson et al. (1) published a seminal work on "Networking Named Content", finally leveraging the information-centric networking paradigm on the research level. The great merit of the paper was to simplify the ideas from DONA and to present a neat and plain architecture, called *Content-Centric Networking* (CCN). The most remarkable change is that the new architecture is a substituent for the Internet layer, including the sacrosanct IP. Information is entirely decoupled from sources. The baseline idea of CCN is to request content directly by name. Users should only have to name what they are looking for and the network will deliver the data. Data can be delivered by the original source server as well as by some intermediate or close-by caches. Names are not only descriptions of the content. They replace the function of addresses. Hence, content is requested, located and delivered directly by name from anywhere in the network. Likewise, all this applies to NDN.

Sender-Driven vs. Receiver-Driven

DONA is like IP *sender driven*. A data source has full control of when and at which rate data is sent to which host(s). Host addresses, present in both IP and DONA, are a requirement for sender driven data transmission. Without them,

the source would not be able to identify and reach requesters. However, the absence of any host addresses in CCN/NDN implies that senders cannot distinguish multiple requesters. This in turn has the direct implication that information retrieval must become *receiver driven*. There is no possibility for a server to spontaneously send data to initial requesters. Accordingly, receiver driven means that there is no data transmission without an explicit request for information. An effect thereof is that the transmission rate is controlled by clients instead of servers.

1) Naming

In CCN/NDN, names are no longer flat cryptographic hashes as in DONA, but rather hierarchical and human-readable *content names*. While both hierarchy and human-readability are not strict requirements, content names are often represented as URL-like content identifiers, composed of multiple name components. The notation that became accepted, i.e. separating name components by forward slashes, also evoke URLs. However, the slashes are not part of the name and used for convenient notation only. Content names are considered at least partly human-readable. An indirection infrastructure may be necessary to map those names to secure names. Hierarchy is mainly mentioned in connection with routing scalability. Hierarchical names can be aggregated due to their structure (e.g. (1), (21), (22)). Regardless of whether the name prefix is hierarchical or not, the prefix should be globally routable. Beyond the globally routable prefix, there can be an arbitrary number of name components with additional semantics for applications and humans. Examples are versioning and segmentation information, timestamps or cryptographic hashes. Instead of segments, often the synonymous term *chunk* is used. Segmentation or chunking is located at the transport layer. Its purpose is to efficiently recover from packet losses. Frame sizes, i.e. payload plus protocol headers, at the transport layer are usually adjusted to carry a frame of the Internet layer which in turn should not exceed the Maximum Transmission Unit (MTU) of Ethernet or another data link layer protocol. This practice guarantees a high level of efficiency at the data link layer. Exposing this feature in content names at the Internet layer allows convenient access for any transport layer protocol. Yet to mention is the literally unlimited address space that is covered by names. Content names can have an arbitrary number of name components with an arbitrary length each. However, the virtually unlimited address space of IPv6 is not inferior to it and should be future proof as well. Figure 2.1 shows an example content name in notational representation.

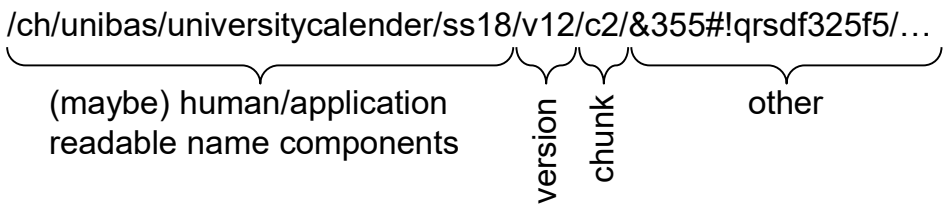


Figure 2.1 – Example Content Name

Possible composition of a content name inspired by (1) in notational representation. The forward slashes delimit individual name components which can have different meanings. Content names are the "addresses" of contents, meaning that an information-centric protocol like CCN or NDN should locate contents directly by name.

In section 6.1, we will present a simple mini language that is inspired by this notation and that will incorporate the forward slashes for syntactic reasons.

2) Name Resolution and Data Routing

Name resolution means the process of finding a data source according to a request. Data routing is the inverse process of finding a path for the data to travel back to the requester. In CCN/NDN, these two processes are coupled. Hence, there is no off-path data routing. Replies always travel back the exact reverse path of the request. Information requests are called *interests* (or interest packet), the replies are called *data packets*. Data packets mainly carry a content name together with the associated data. Further data packet fields are discussed as necessary. Interests primarily carry a content name. NDN interests additionally contain a *nonce*. The nonce is a random value that is used to detect looping and multiple copies of the same interest. They should be discarded so that data packets are returned only over one return path instead of over every possible path. A list of recently seen nonce values must be either maintained separately or stored together with PIT entries. In contrast, CCN(x, since version 1.0 (23)) does not have nonce values in interests and uses hop limits to prevent interests from looping.

An inbound interest is first matched against the *content store* (CS). The CS is an in-network and on-patch temporary cache. The caches store data that recently has travelled over the node. It is up to the node administrator to choose a metric that decides which content is kept longer or dropped earlier from the cache. If there is a content in the CS that matches the interest, a copy of the data is sent back to the requester over the same interface the interest came from. We call the interface on which an interest arrived *arrival interface* (AI). Content matching is important for this thesis and is therefore examined in a separate subsection 3.3.2.

If there is no match in the CS, the *pending interest table* (PIT) is checked for a match. The PIT temporarily stores all interests that have been forwarded, together with a list of arrival interfaces over which the interests entered the node. An interest that travels from a requester towards one or more data sources thereby creates one or more traces through the PITs of all visited nodes. The PIT has three tasks: First, it aggregates interests for the same information that has already been sent upstream once. If the interest came from an interface that is not yet in the list of arrival interfaces for that particular interest, the interface is appended to the list of arrival interfaces. However, the interest is explicitly not forwarded again. Not having to pass every individual request through to the data source is one of the attractive design considerations of CCN/NDN. Second, the PIT should resend interests that remained unsatisfied for a predefined amount of time. Third, the PIT is responsible for *data routing*. This is, to distribute incoming data packets back to all interfaces in the according list of arrival interfaces. A node only knows the next downstream node for a data packet, respectively over which interface to reach that node. From a global perspective, a data packet exactly follows back the trace that was created during the interest's upstream journey. Thus, CCN/NDN only support on-path data routing, i.e. coupled name resolution and data routing. Concerning interests, PIT matches must be *exact* matches. Interests are aggregated only if an interest with the exact same name (not including the nonce) has already been forwarded. The approach with a separate list of recently seen nonce values is preferable because names of aggregated interests

must then be stored only once in the PIT. Matching of data packets to existing PIT entries is also examined in subsection 3.3.2.

If there is no match in the PIT, a new PIT entry is installed, and the interest is guided towards the content with the help of the *forwarding information base* (FIB). FIBs contain mappings from name prefixes to interfaces. Forwarding decisions are made through matching, too. There can be several matching prefixes in the FIB, however, only the *longest prefix match* (LPM) will be considered. The number of interfaces associated with a prefix can be greater than 1 as the information might be retrievable in several locations. The original approach foresees to forward the interest to all interfaces associated with the LPM. This inherent multipath routing is another attractive feature of CCN/NDN. Nevertheless, operators may resort to any kind of rule or heuristic to limit the number of forwarding interfaces. According to the original proposal by Jacobson et al. (1), an interest is discarded if there is no match. Compagno et al. (24) alternatively propose to answer stranded interests[2] with a negative acknowledgment (NACK), containing additional reasons of failure. CCN and NDN protocols are, like the IP, not responsible for gathering routing information. They only perform forwarding based on content names and node-local routing information.

3) Caching

On-path caches are, unlike DONA, no longer a voluntary extension but integrated constituent of the architecture. As mentioned above, content stores take over this role in CCN/NDN. The universal caching approach is especially helpful in scenarios with popular, i.e. frequently requested, content. Local bursts of equal requests then only affect a small part of the network, between eruption center and closest cache. The rest of the network will not even notice the situation. However, it is obvious that such "buffer memories" (1) are not able to cache the whole traffic that went through the nodes. Psaras et al. (25) calculated that a 10 GB cache can keep track of the traffic over a 1.2 Gbps link for roughly ~64 seconds down to ~0.5 seconds of the traffic over a 159.2 Gbps link. Hence, content stores do not serve as long-term storages, requiring an adequate data replacement strategy. Jacobson et al. (1) mentioned *least recently used* (LRU) and *least frequently used* (LFU). Every data replacement strategy has the goal to improve the *cache hit ratio*. Both LRU and LFU pursue this goal in a very confined single node perspective. However, what rather should be of interest is the cache hit ratio along a full request path, or more generally speaking, in a whole network. If every node along a path implements LRU or LFU, the network will suffer from heavy caching redundancy. Chai et al. (26) showed that even random caching at a *single* node along the delivery path is similarly successful as LRU. The performance gets even better if every node along the path caches content randomly. The effect is due to increased diversity, i.e. decreased redundancy, of data. This insight motivated a centrality-based caching algorithm with superior performance. It bases on a per-node measure called *betweenness centrality*. The measure gives evidence about how often a node is part of the content delivery path between every pair of nodes in a network topology (26). Only the node with highest betweenness centrality caches the data. Furthermore, Psaras et al. (25) introduced a caching algorithm that makes probabilistic caching choices. They observed reduced cache-eviction by an order of magnitude which let us suggest an increased data diversity, too.

[2] interest that could not be satisfied or with lacking forwarding information

By 2017, the International Telecommunication Union (ITU) estimated 4.22 billion active mobile-broadband subscriptions worldwide, resulting in 56.4 subscriptions per 100 inhabitants worldwide and 97.1 subscriptions per 100 inhabitants in developed regions (27). Subscriptions in developing countries are assumed to further increase, not least because the rise of newer technologies like wearables and connected cars. Therefore, appropriate support for host mobility is an essential requirement for every new network layer approach. Support of mobility has two aspects: *consumer mobility* and *producer mobility*. As discussed above, IP does not support either of them very well. Whenever the consumer or the producer moves to a new network location, they need to acquire a new IP address. Subsequently, the communication partner must be notified about the new address. For example, the Host Identity Protocol (HIP, (28)) features a rendezvous server (RVS) for such notifications. Not having any concept of host addressing, CCN/NDN does not need to inform communication partners. Note that this would not even be possible without overlaying coordination.

Consumer mobility is perfectly supported in CCN/NDN. A moving consumer only needs to resend every unsatisfied interest after changing the network location. As previous interests might already have attracted the requested content to a close-by cache, the latest interest is likely to be quickly satisfied with a small impact on the overall network.

Producer mobility is trickier. Basically, all routing information concerning the dislocated producer must be updated. However, this approach will not scale to the Internet. HIP-like solutions, where producers keep a RVS up-to-date about their current location, are conceivable. Afanasyev et al. (29) presented a *mapping* approach where the RVS returns a globally routed[3] prefix corresponding to the content name. The name prefix can be used by the requester to guide the interest towards an intermediary that should know the current producer location. This approach requires the globally routed prefixes to remain static. In contrast to mapping, *tracing* means that the RVS serves as an indirection point for request and reply. For example, Hermans et al. (30) described a solution where such an indirection point encapsulates the original interest in a new interest with a temporary source prefix. These encapsulated interests are then sent after the moving source. Another paper from Hermans et al. (31) describes an encapsulation-free approach that works with locator/identifier split (LISP). Interests are equipped with an optional location name field that helps to forward interests towards content. The indirection point can either deliver the location name back to the requester, which then must issue a second interest, or the indirection point forwards the equipped interest by itself. Zhang et al. (32) summarize these approaches as "mobile producer chasing". Furthermore, they point to another class of approaches called "data rendezvous". The idea thereof is to either shift data produced by a mobile host to a stationary (rendezvous) server ("data depot") or to enforce that data is 'produced within' and 'available from' a stationary region ("data spot"). In both cases, the mobile producer leaves the custody over its data to third. In case of sensitive data, this might cause some security concerns. They are treated next.

[3] and routable, note the difference

5) Security

We distinguish the terms 'securing data' and 'encrypting data'. Securing data means the process of establishing bindings such that the consumer can verify the *integrity* and *provenance* of data. CCN/NDN achieves this by binding the content name to data through a *digital signature* (1). Concretely, the producer creates a hash over name and content and encrypts the digest with its private key. Every data packet must contain such a signature together with information about the used hashing algorithm and either the public key of the producer, i.e. the signer, or information where this public key can be retrieved (1), (20). Jacobson et al. (1) calls the latter *signed info*. To verify data integrity, the requester must also create the hash over name and content. After decrypting the signature with the public key, the two digests can then be compared to each other. To verify data provenance, the requester must trust the owner of the public key. This trust can be established trough a binding between public key and name or public key and producer. Such bindings can be established by Validation Authorities (VAs). In the former case, a VA must confirm that the owner of the public key is allowed to sign the given name. In the latter case, a VA must reveal the owner of the public key such that the requester can check whether the key owner is the expected producer or not. Apart from (commercial) third-parties, every user-defined trust anchor can serve as VA.

Self-certifying names are also possible in CCN/NDN. They might be useful to name content such that the name itself does not allow to draw conclusions to the content. However, self-certifying names require an indirection infrastructure to map between them and human-readable names. According to Jacobson et al. (1), such indirection infrastructures are often another security risk as they are unsecured.

Signed data allows users to accept data packets no matter where from they were delivered. A valid and trustworthy signature guarantees innocuousness. Producers can encrypt data on a packet- or chunk-wise base. This allows them to disseminate sensitive data. Encrypting data enables *confidential* data exchange, a third pillar in security. In today's Internet, the normal case is that a server stores unencrypted data. Restricted access can be enforced through client authentication. In addition, a secure connection between client and server, e.g. established through the Transport Layer Security (TLS) protocol, ensures that no one else can read the information. However, channel-based encryption is rather coarse-grained, compared to CCN/NDN's packet- or chunk-wise encryption.

Both *asymmetric* and *symmetric* encryption is applicable in CCN/NDN. If requesters are known, data producers can use their public keys to encrypt data. This is the asymmetric case. Data packets can be released carefree to the network. Only requesters can read the data by decrypting the messages with their private keys. A drawback of asymmetric encryption is that data must be encrypted individually for every distinct user. This implies that data is encrypted only on-demand. Encrypting all data for all known users in an anticipatory way is likely to be too expensive. The symmetric case is more suitable for anticipatory encryption because data needs to be encrypted only once. Symmetric encryption enables to place encrypted data in caches even before the content is requested. However, if requesters are not already in possession of the symmetric key, they must first request it. To exchange keys securely, the communication partners may again resort to asymmetric encryption.

2.2.1.1 Immutability

An important design principle of both CCN and NDN is that data packets are designed to be *immutable*. The term is not restricted to a specific field such as computer science. However, it is commonly used in object-oriented programming (OOP). Two severities are differentiated: *Strong* immutability means that not a single field or component of an object can be changed after its instantiation. *Weak* immutability is given when some fields of the object may change while others must not change. For example, C++ uses the *const* type qualifier (33) to indicate immutable fields, i.e. constants. Java uses the keyword *final* (34) while C# differentiates between *readonly* (35) for run-time constants and *const* (36) for compile-time constants. If not specified explicitly, immutability is normally understood as strong immutability. Broadening this thought over data packets, strong immutability means that not a single bit can change once the data packet was created, not in the payload nor in any other field.

Imposing such a strict rule seems implausible at first. However, immutability is a requirement for an effortless implementation of universal caching. When being sure about the unchangeability of data packets, every data packet can unconcernedly be cached everywhere and multiple times. Immutability prevents cached copies to get in an *inconsistent state*, i.e. that differing instances of the originally same packet may be present in different caches. Thus, there is no need to keep track of all copies and update them continuously to re-establish consistency. Inconsistent cache states are to be avoided as they lead to the undesirable situation where two users requesting the same content will be served with differing data packets.

However, it ultimately is impossible to make data packets physically immutable in a context like the Internet. Despite this, fraudulent manipulations can be detected, e.g. with the signing mechanism that checks data integrity as described in the above security discussion. Strictly applied, this prevents inconsistent cache states, too.

2.2.2 Generalized CCN/NDN Execution Model

So far, we only described protocol mechanisms in written form. To summarize CCN and NDN's generalized execution model, Figure 2.2 illustrates the most important steps. The legend is given in the caption. The figure also serves as reference for the extended execution model in section 5.5.

Note that a node does not actively switch to a waiting state after an interest has been forwarded (see marker A in Figure 2.2). A node always listens to incoming data packets, irrespective of an interest being forwarded in advance or not. Likewise, data packets are treated always the same, no matter if they were once requested or not. If there is a matching PIT entry, a data packet is returned to all requesting faces. Otherwise, it is dropped. The way of representation was chosen to disambiguate that unsatisfied interests can be re-polled periodically. Even requested data can be dropped, namely when the request has been satisfied previously from another source. Unless some sort of path selection is applied, this can occur quite frequently due to CCN and NDN's intrinsic multi-path routing.

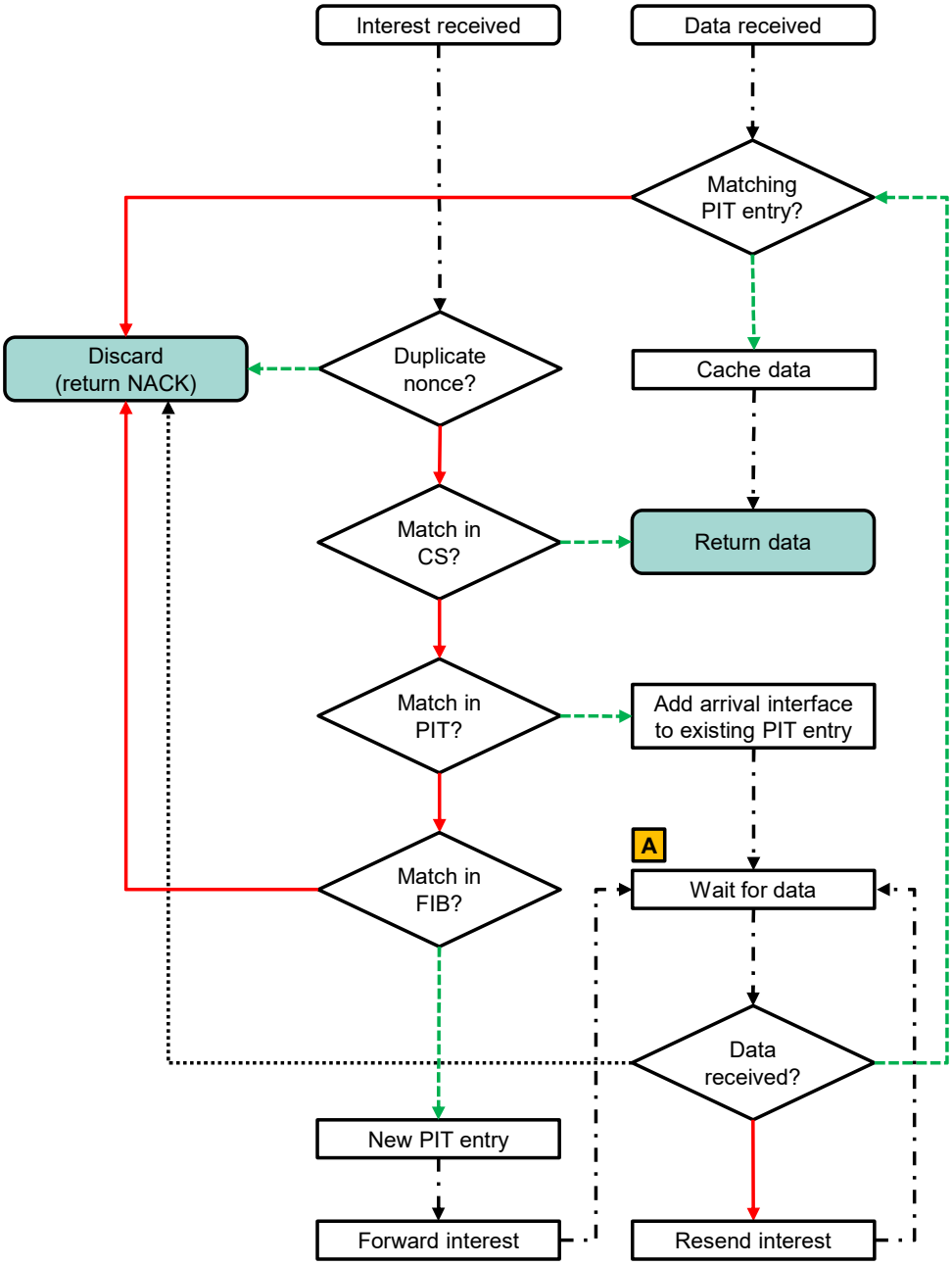
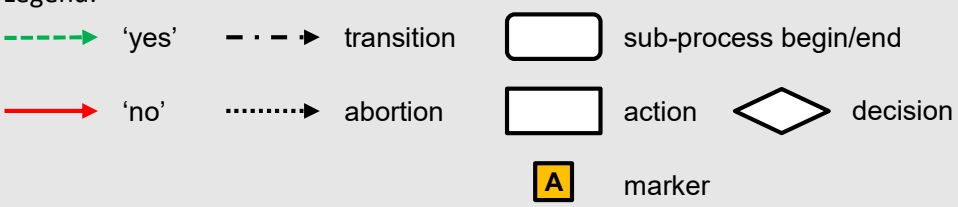


Figure 2.2 – Generalized CCN/NDN Execution Model

In this flow chart, boxes with round corners indicate beginning or ending of a sub-process while rectangles are actions. Rhombs indicate decisions. Arrows are flow lines. A dashed (green) arrow coming from a rhomb indicates 'yes' while a solid (red) arrow is 'no'. Morse code (black) arrows are transitions. Dotted (black) flow lines indicate process abortion.

Legend:



2.3 Named Function Networking

Tschudin and Sifalakis (3) proposed a generalized approach called Named Function Networking (NFN). The baseline idea of NFN is to ask for results of functions instead of named content. They argue that preliminarily produced information is just a special case. In contrast, NFN enables users to request complex, tailor-made and on-demand results. An interest is rather an expression than only a name, composed of function calls and their arguments. Arguments can be yet another expression with further function calls or values. Name resolution turns over to expression resolution. The authors decided to restrict expressions at the top level to *lambda-expressions*. λ-expressions adhere to the rules of λ-calculus, a formal and Turing complete model to express computations. The core of the model are the three components *variable*, *abstraction* and *application*. All three components are itself valid λ-expressions.

λ-Calculus

- **Variables** are named placeholders. They are used to build up abstractions. They finally will be substituted by values during the evaluation of expressions.
- **λ-abstractions** are anonymous function definitions. Anonymous means that no name is attributed to the function. They consist of exactly one input variable, i.e. a parameter, and a second λ-expression. This second λ-expression shows where the input variable occurs, respectively where the input variable must be replaced by the actual value. Functions with more than one parameter can always be transformed to a chain of functions with one parameter each. This is known to as *currying*. For currying to work, it is required that functions are so-called first-class members, meaning that functions can be both input and output of a function. λ-calculus fulfills this requirement because every expression is a λ-expression.
- Finally, the **application** is the invocation of a λ-abstraction, i.e. a function call. Thus, an application also consists of two expressions, namely a function definition and the input.

Evaluation

Evaluation of a λ-expression comprises consecutive reduction of subexpressions. Reducing subexpressions means to replace original subexpressions with equivalent but simplified expressions. As soon as no further simplification is possible, the result of the computation is found and can be obtained. Moreover, λ-calculus has one appealing feature that is described by the *Church-Rosser theorem*. The core of the theorem says that no matter how the order of evaluation in alternated, the final expression is the same (37). Reduction rules with this property are also said to be *confluent*. This is particularly interesting in terms of distributed computing. Subexpressions can be evaluated in any order as they are independent of each other. Larger expressions can be split in smaller subexpressions and distributed over a network, without the need to coordinate their execution. The final result will be the same. Sifalakis et al. (38) allude to different evaluation strategies facilitated by λ-calculus. Especially *call-by-name* is highlighted. It enables to delay the evaluation of argument expressions to the moment where the argument is used. This is achieved through replacing all occurrences of the input variable trough the unevaluated argument expression. However, in terms of network traffic, this strategy is only beneficial in cases where the evaluated expression is larger in size than the unevaluated expression. Contrariwise, the network traffic

is increased. In terms of processing resources, delaying the evaluation is only beneficial in cases where the parameter does not occur at all or in cases where a computation is aborted before termination. Both cases should eventuate seldom. Whenever the parameter occurs more than once, the computational effort is multiplied by the number of occurrences.

2.3.1 Implementation

Although NFN is as generalization of CCN/NDN, the *resolution engine* is implemented on top of CCN/NDN. The core of the engine is a slightly extended Krivine Abstract Machine (39), performing call-by-name λ-expression resolution. The resolution engine is designed as individual component within normal CCN/NDN nodes. Interests that should be interpreted as λ-expressions contain an implicit postfix name component `/NFN`. Normal CCN/NDN nodes, i.e. those without a resolution engine, simply forward the expressions according to LPM. The implicit postfix name component has no direct effect. However, NFN nodes realize the name component and hand over the λ-expression to the resolution engine. The request pattern for static content remains unchanged. This practice allows to fully harmonize with normal CCN/NDN nodes.

Computation Configuration

NFN's abstract machine works with so-called *computation configurations* (3). They consist of four elements: 1) name of an environment, 2) name of an argument stack, 3) expression and 4) name of a result stack. 1), 2) and 4) are all implemented as CCN/NDN-like named data. Remember that λ-calculus variables are named placeholders that will be substituted later by values. The environment is an association table from bound variables to closure names. A closure associates the environment with an expression. This way, a variable can be looked up in a specific environment, to then be replaced by the corresponding expression. The argument stack is used to push and pop closures during the evaluation of an expression. It reflects the intermediate state of the computation. The result stack offers an additional location to push and pop operands during the application of operations.

Whenever a λ-expression is evaluated, the abstract machine continuously takes environment, argument stack and result stack and creates updated copies of them, according to the instructions found in the expression. New and updated state copies are published, i.e. made accessible, under new and unique names. Hence, whenever a state needs to be changed, a new copy will be generated. Note that these copies are truly new content and not only duplications as they occur when a content store satisfies an interest. For example, the simplest action of resolving a variable by name involves two updated copies of the argument stack (one push and one pop). λ-abstractions and applications incur even more copies.

The reason for creating copies instead of directly modifying stacks and environment is that NFN strives for compatibility with CCN/NDN. NFN-enabled nodes should be able to coexist with nodes that only expose CCN/NDN capabilities, albeit NFN is a generalization of CCN/NDN, and not only an extension. Accordingly, NFN must comply with CCN/NDN characteristics like the immutability property of its data objects. Because stacks, environment and even computation configurations are implemented as such immutable data objects, it is crucial not to modify them directly.

FOX Instruction

This indirect approach with an abstract machine has one major advantage. Complying with the immutability property guarantees that no old state can be lost or destroyed through overwriting. Making every successfully completed computation step persistently available, imply that aborted computations can be resumed exactly where they have been interrupted. The *find-or-execute* (FOX) primitive first searches for an (intermediate) result of a computation before its re-computation is triggered again (3). For FOX to work, the *computation configuration after reduction* needs to be stored under the *canonical hash over the initial computation configuration*. The computation configuration after reduction says where the (intermediate) result can be found, i.e. on top of which result stack.

2.3.2 Expression Rewriting

In CCN/NDN, FIB states implicitly define the paths an interest will take. Hence, a requester has no possibility to directly influence the resolution of a request. Despite this, NFN provides the opportunity to take influence on the distribution of computations. The opportunity arises from the way how NFN makes use of CCN/NDN's infrastructure. To send named functions over a CCN/NDN, λ-expressions must be mapped to content names. The default mapping is inversion, hence λ-expression $\lambda_{2.1}$

```
λ2.1: /Name/Of/OuterFunction(
            /Name/Of/InnerFunction(/Name/Of/Data))
```

is inverted to interest $i_{2.1}$

```
i2.1: /Name/Of/Data | /Name/Of/InnerFunction |
            /Name/Of/OuterFunction
```

where the pipe character '|' is used to indicate the name components in the content name. Thus, $i_{2.1}$ would be forwarded in direction of `/Name/Of/Data` first. Note that the system's priority should be to satisfy $i_{2.1}$ with a single final result, not three partial results. If this undertaking was not successful, there is still the possibility to find partial results, i.e. results of subexpressions. The initial requester can search for them by consecutively tail-drop name components. Due to inversion, the shortened content name corresponds to interest $i_{2.2}$.

```
i2.2: /Name/Of/Data | /Name/Of/InnerFunction
```

(Re-)computing an expression should be considered only if no partial results were found. In this case, subexpressions (data and functions) can be fetched individually. For example, the provider of `/Name/Of/Data` can request the code of `/Name/Of/InnerFunction` and `/Name/Of/OuterFunction`. On successful retrieval, the data provider can consecutively apply the inner and outer function calls to the data.

However, in λ-calculus it is possible to *rewrite* λ-expression into equivalent λ-expressions. For example[4], $\lambda_{2.1}$ from above can be rewritten to the equivalent $\lambda_{2.2}$.

[4] in dependence on examples in (38), p. 139

```
λ2.2: (λxy.(x (y /Name/Of/Data))) /Name/Of/OuterFunction
            /Name/Of/InnerFunction
```

Mapping this rewritten expression $\lambda_{2.2}$ to a content name results in interest $i_{2.3}$.

```
i2.3: /Name/Of/InnerFunction | /Name/Of/OuterFunction
            | (λxy.x y /Name/Of/Data)
```

From the point of view of λ-calculus, there is no benefit to do so as both expressions will resolve to the same result. Nevertheless, it influences the name resolution process in CCN/NDN. Obviously, $i_{2.3}$ will be forwarded first in direction of `/Name/Of/InnerFunction`. NFN nodes should always be clear on the ordering of expressions. Consider the following example: Tail-dropping a name component in $i_{2.3}$ corresponds to crossing out the inner-most expression. What remains is a request for `/Name/Of/InnerFunction | /Name/Of/OuterFunction`. Without being clear on ordering, this could erroneously be understood as request for the composition function

```
   (InnerFunction ∘ OuterFunction)(Data)
=  InnerFunction [OuterFunction(Data)]
```

This call is no longer equivalent to the original λ-expression. However, if being clear on ordering, the ability to express the same expression in equivalent forms, bears great opportunities for users to express their preferences on how expressions *should* be resolved. Note that the final resolution path cannot be controlled entirely by the requester. Every involved NFN node can rewrite the requests according to network conditions or just at whim. Sifalakis et al. (38) called this ability *preferential opportunism in the distribution of computations*. Beyond CCN/NDN-like pulling of code or content and making use of cached results, they also demonstrated how to push computations from one NFN node to another. This can be particularly helpful to discover new paths and for implicit load-balancing.

Having a conceptual approach to offload computations that leaves room for versatile optimizations sounds promising. However, with *cloud computing services*, an established technique for the delegation of computations exists. Also, newer computing phenomena like *edge* and *fog computing* are spreading, requiring to answer the question why they might be beneficial compared to established techniques. The upcoming section will lead through those topics, discussing pros and cons and clarifying why NFN elegantly matches edge and fog computing fundamentals.

2.4 Cloud & Post Cloud Computing

Shifting computations from machine to machine has been done ever since the rise of computers. Even before classic personal computers (PCs) penetrated mass market, a widely-spread way of doing computations was the mainframe. Their main characteristics are high computational speed and high *reliability* and *availability*. Computations were issued from interactive user terminals to the mainframe. As the main load is carried by the mainframe only, the approach is

super-centralized. Centralization also applies to storage. Data is stored in mainframes. Terminals only display information.

In contrast, the boom of PCs has been a true step in direction of *decentralization*. Both computational power and data storage laid now in the hands of individual users. This development is tightly connected with price decline of necessary hardware and the coupled marketing in the sense of "have your own mainframe". However, by means of reliability, this development was at first not necessarily for the benefit of users. Mishandling and technical deficiencies of soft- and hardware were a permanent source of data loss. Furthermore, the availability of PC systems was lower compared to mainframes because detecting, repairing or replacing failed components was rather cumbersome and not done by experts. However, steady improvements in reliability (e.g. CPUs rarely fail) and availability (e.g. affordable RAID configurations) over the decades pursued the success story of PCs.

Nevertheless, it was not solely due to PC's immobility and continuously increased user mobility that necessitated novel computing architectures. Notebooks and nowadays smartphones are mobile, fast and have plenty of storage capacity for everyday use cases. The reasons must be searched elsewhere.

2.4.1 Cloud Computing

End user devices such as smartphones and tablets massively spread in the last decade. In the developed world, it is not unusual to have a PC/notebook at home, another one at work plus a smartphone and a tablet computer, maybe even further connected devices such as smart TVs or network-attached storages (NAS). Nevertheless, users generally have just one notion of how to organize their information. It is hard to remember which files, favorites, e-mails etc. have been stored on which device. Users expect to foster their ecosystem of information just once and to be able to access it from any device. Moreover, users expect to produce and access their information *en route*. Thus, consumer and producer mobility, as discussed above, are again of interest.

Remote Services

For e-mailing, the problem of synchronization was diagnosed more than 20 years ago. Crispin (40) proposed in 1994 a protocol that "allows a client to access and manipulate electronic mail messages on a server" (40) and furthermore "provides the capability for an offline client to resynchronize with the server" (40). The protocol was named Internet Message Access Protocol (IMAP). The basic idea was to maintain up-to-date versions of user mailboxes ("remote message folders") on a server. Client terminals synchronize their local copies of the mailboxes to the server's state. This contrasted with the Post Office Protocol (POP, (41)) that foresees to download and delete single e-mails. Hence, IMAP was a great leap forward in terms of usability and a strong hint what users expect: services that are accessible from *anywhere* and *at any time*.

Cloud Services

Over the years, more and more services were relocated to servers in large data centers from where they were served to the clients. Common examples are a) provision of storage, b) computational resources such as CPUs and RAM, c) run-time and development environments and d) applications. Well known service providers and/or products are for a) Dropbox[5], Microsoft OneDrive[6],

[5] https://www.dropbox.com
[6] https://onedrive.live.com/about/en-us/

Amazon Simple Storage Service (S3)[7], for b) Amazon Elastic Compute Cloud (EC2)[8] for c) Microsoft Azure[9] and Google App Engine[10] and for d) Microsoft Office 365[11]. For better differentiation of service types, the products were classified into the three groups *infrastructure as a service* (IaaS), *platform as a service* (PaaS) and *software as a service* (SaaS). a) + b) are representatives of IaaS, c) of PaaS and d) of SaaS. All service types together are summarized as *cloud computing* services. Setting them apart from the decentralized PC age, cloud computing services have some attractive commonalities. They are *location independent*, i.e. accessible from everywhere. For example, a service like Dropbox implements the data depot case discussed in 2.2.1/Mobility. (Mobile) producers can upload their data, e.g. to keep their travelling blogs updated, while other (mobile) users can access the data, e.g. read the blogs. However, Dropbox is at the application layer and therefore cannot offer network layer functionality like in-network caches. It also does not allow to implement its own upload protocol that does not rely on third parties (32). Back to the advantages, cloud computing services are normally *device independent*, meaning that only a browser is needed to access them. Access to data and resources can be shared through access control. Generally, the services scale well. Clients can adjust resources on-demand. Services are flexible, allowing fast acquiring and release. Cloud service providers can improve the utilization of their infrastructure. Dynamic resource allocation can drastically reduce idle time compared to dedicated hardware, enabling to make well-priced offers to clients. Finally, cloud services are an extra layer of security. Data replication and up-to-date security measures protect against loss of data. Redundant hardware ensures business continuity and a higher degree of (distributed) denial-of-service ((D)DoS) resilience.

Reservations

In this sense, implementations of IMAP were early cloud computing services. Nevertheless, in opposition to IMAP that is a proposed standard (40), free to be implemented for everyone, nowadays cloud computing services are rather vendor specific applications, relying on established network technologies like IP, (secure) hypertext transfer protocol (HTTP(S)) or the file transfer protocol (FTP). The focus does not lie on creating a networking standard but on commercialization by binding users to a specific ecosystem. Furthermore, a distinct tendency towards *everything as a service* (XaaS) can be observed. Users are invited to shift more and more tasks to the cloud, preferably always to the same ecosystem, increasing the dependence from users on vendors. As a consequence, client devices (terminals) degenerate to displays with a network connection. Zhou et al. (42) described this evolution as "a historical and spiral regression to the centralized mainframe computing paradigm". Indeed, data centers itself are highly optimized in every sense but the (possible) degree of centralization reached such an extent that resources are again wasted in end devices, lowering the overall efficiency of the cloud (that includes edge devices).

Cloud computing services are a nightmare in terms of *privacy*. None of the big cloud storage provider offers built-in client-side encryption. Willingly, the enforcement of the 'code of conduct' is cited as reason. However, knowledge about user activities are clearly of business interest. Even if data is stored

[7] https://aws.amazon.com/s3/
[8] https://aws.amazon.com/ec2/
[9] https://azure.microsoft.com
[10] https://appengine.google.com
[11] https://www.office.com/

encrypted, users potentially disclose many private information like hours of work, place of work, itineraries, contacts, etc.

Furthermore, cloud computing has a negative impact on network load. Instead of doing computations itself, users send their data to a remote data center where it is processed. Not infrequently, the same users then request the result. Not infrequently as well, producers and consumers are geographically close to each other, e.g. relatives, colleagues at work or a home computer to a workplace computer. On the other hand, the bypassing data center that provides the always-on intermediary service might not even be on the same continent[12]. This can cause large amounts of traffic over long distances. Longer distances usually come along with higher *latencies*. This is especially a challenge for interactive applications that require low-latency access to results. Zhou et al. (42) mention gaming, video streaming and augmented reality (AR) as examples. These are not dreams of the future only. Offers like GeForce NOW[13] where games are running in a server farm, i.e. in the cloud, are yet reality. The given example also demonstrates that providing enough processing power is not the limiting factor. The bottleneck is the network. Reactions to user inputs, e.g. steering commands, should be available in real-time. Given enough bandwidth and assuming 120 frames per second (fps) for a smooth gaming experience, real-time reactions defined to as user inputs taking effect within one frame require a round-trip time (RTT) $\leq 1/120$ s ≈ 0.00833 s ≈ 8 milliseconds. This is hard to achieve even when neglecting game processing time. Improving the latency for a given path is incredibly challenging as network routing and packet processing are already extremely optimized.

Along with the rise of the *Internet of Things* (IoT), additional requirements arose. The cloud computing approach is not fully able to meet many of them.

2.4.2 Internet of Things (IoT)

The Internet of Things (IoT) is understood to be the interconnection of things and things to the Internet. A thing can literally be anything that is able to communicate. There are no limitations to the communication technology or the capabilities of the thing. Thus, a thing can be anything between sensors/actuators and data centers. What they have in common is their involvement in the creation and/or processing of information plus their ability to exchange information.

Compared to the cloud, what is new in the IoT is the sheer number of things, especially driven by large quantities of available low-cost things. There are many estimations on concrete numbers and prognoses. To give a conservative estimation, Gartner (43) forecasted in February 2017 a total of 8.4 billion IoT devices for 2017, 11.2 billion for 2018, and 20.4 billion for 2020 (not including mobile phone, PC/laptop/tables, fixed phones). Ericsson (44) estimated in its November 2015 mobility report to see 28.2 billion devices in 2021 (including 12.9 billion mobile phones, PC/laptop/tables, fixed phones) and the International Data Corporation (IDC, (45)) assumes 30 billion connected things for 2020.

Scenarios

Example scenarios for the application of the IoT are almost innumerable. Often mentioned are sensor networks to observe indicators like traffic volume

[12] For example, AWS does not yet maintain any infrastructure in Africa (110).
[13] https://www.nvidia.com/en-us/geforce/products/geforce-now/mac-pc/

(smart city), insolation or temperature (smart home), traffic situation (connected driving), blood glucose, blood pressure, pulse (healthcare), or electric power consumption, voltage and amperage (smart grid). Nevertheless, it is not necessary to have a conglomerate of sensors to be classified as IoT. Single device scenarios, as for instance activity tracking with wearables, are also covered by the IoT. Furthermore, the IoT is not only about *sensing*, *gathering* and *production* of information. It also includes to *react*, *steer* and *process* information. For example, if a smart home measures intense insolation on windows, it is desirable that the smart home activates an actuator that lowers the blinds. Another example is vehicle to vehicle (V2V) communication where a vehicle gathers information from and about surrounding vehicles, reacts according to the information, e.g. accelerates, and further propagates the information to other vehicles.

Challenges

The reasons for the outstanding challenges of the IoT are the amount of data things produce, their low storage and processing capabilities, and their heterogeneity. Sending all captured information to a distant data center is unfeasible. Cisco forecast in their Global Cloud Index 2016-2021 white paper (46) the total *amount of data* created by all devices to grow from 218 ZB[14] in 2016 to 847 ZB in 2018, mainly caused by the IoT. Comparing these numbers to the annual global IP traffic of 1.2 ZB in 2016 and an estimate 3.3 ZB in 2021 (47) bode ill. Already a very small portion of the produced data being sent to data centers would let collapse the Internet, assuming that the available infrastructure limits the global IP traffic. Hence, there is no other choice than processing data near to where it was produced and to only send advisedly selected or summarized data to the cloud. Reasons may be historical analysis or long-term storage (48).

However, the *low storage and processing capabilities* of things require an *efficient distribution of content* as well as *offloading of computations*. Remember that the cloud computing approach does not suffer from low storage or processing capabilities, but it lacks to satisfy time-sensitive applications. An example could be traffic analysis and the question where the tail end of a traffic jam is. Sensed data may be expired before it was processed. Thus, any result found from that data may be useless or even dangerous.

A further challenge is the *heterogeneity* of things. Any kind of IoT device should be able to communicate with each other, with gateway devices at the edge and even with cloud infrastructure. This contrasts with the cloud that is rather homogenous. Hard- and software is under control of the specific vendors, enabling them to put together optimally matching components. Heterogeneity complicates the development of standards that fit for any kind of IoT device and scenario. The lack of standards in turn hampers the efficient utilization of resources because every vendor or service provider only pursues self-optimization instead of collaboration.

Whether it is efficient infrastructure usage or to avoid the collapse thereof, both calls for a new way of involving network attached devices in service provision. This is what post-cloud computing[15] approaches like edge and fog computing promise.

[14] 1 zettabyte = 1 billion terabytes
[15] term taken and reused from (42)

2.4.3 Edge and Fog Computing

Both edge and fog computing are not sharply defined terms and merge with each other. They are not scientific concepts but originate from two different platforms, driven by multiple stakeholders.

Fog

Fog computing is a term coined by Cisco. It first appeared at a talk in 2011 (49) and 2012 in a paper of Bonomi et al. (50) where is was described as "the provision of storage and computing resources, combined with networking services *between* end devices and cloud computing data centers". Another descriptive definition of fog computing is used in the title of another Cisco white paper from 2015 (48): "Extend the cloud to where the things are". In a nutshell, fog computing can be considered as *providing cloud functionality closer to where information is created*. Bonomi et al. (50) only describe the vision and the key characteristics of the platform. A concrete implementation is not presented. They rather address shortcomings of the cloud and challenges of the IoT as we discussed them above. Under the leadership of Cisco, the OpenFog Consortium[16] was created, targeting to make fog computing a concept rather than a product or a specific platform.

Edge

The platform behind edge computing is called *mobile edge computing* (MEC), being proposed by the European Telecommunications Standards Institute (ETSI) in 2014 (51). MEC focuses stronger on the near-mobile-device environment that is characterized by proximity, ultra-low latency, high bandwidth, real-time access to network information, and location awareness. Thus, the platform is intended to run on MEC servers that are directly attached to mobile phone base stations. Feeling that the limitation to mobile radio technologies was too restrictive, the platform was renamed to *multi-access edge computing* (still MEC) in 2016 (42), clarifying that all types of access technologies are supported, e.g. Wi-Fi and fixed-access. Furthermore, the platform is drafted to serve commercial intents. Through attached MEC servers, operators of base stations can grant access to their infrastructure. Customers can then use the platform to deploy their own apps and services. The platform should ensure an improved quality of experience (QoE) and an increased efficiency, creating business benefits like flexibility and agility (52). Accordingly, MEC can be considered as distributed PaaS that *provides cloud functionality closer to where information is created*, similar to fog computing.

However, as initially mentioned, the definitions of fog and edge overlap. Fog computing does not exclude direct data processing on edge devices or gateways as well as edge computing is not limited to end devices. Both approaches place additional hardware between end devices and data centers that is responsible for tasks beyond networking. It is therefore not surprising that Cisco itself has not a clear notion of the two terms. Once they name fog the *standard* and edge the *concept* (53) while in another place, they equate both terms (48).

Pros and Cons

In this thesis, we will concentrate on the term 'fog computing' and define it to cover the whole range from end device to data center, including both ends. Its benefits are scalability, low latency and protection of network resources against data flood produced by the IoT. In addition, fog computing has some attractive security features. As it is no longer necessary to send all data throughout the

[16] https://www.openfogconsortium.org/

Internet, it is hard for men-in-the-middle to intercept or analyze traffic. Privacy of data that still needs to be sent to the cloud, can be improved by local preprocessing, e.g. through anonymization or aggregation. The lower dependence on third party could service providers also has a positive privacy effect. Moreover, owner-operated infrastructure has much lower acquisition costs in case of fog compared to cloud.

Decentralized setups provide high availability through redundancy, even with unreliable hardware. If one site fails, e.g. through fire, another site will still be available. However, the effort to bring up the failed site, especially when physical intervention is needed, is much higher compared to data centers. Modern data centers use sensors for the detection of defective hardware and robots for automated exchange thereof. Hence, the maintainability of fog and IoT infrastructure is inferior to that of cloud infrastructure.

Data centers are also hard to beat in terms of cost efficiency as they are often located where land and power supply are cheap. Their homogeneous hard- and software further simplifies the maintainability, again decreasing the costs. Finally, it is easier to defend centralized and homogeneous setups against attackers. Keeping security related components up-to-date causes less effort. Figure 2.3 summarizes important characteristics and differences of cloud computing, fog computing ant the Internet of Things.

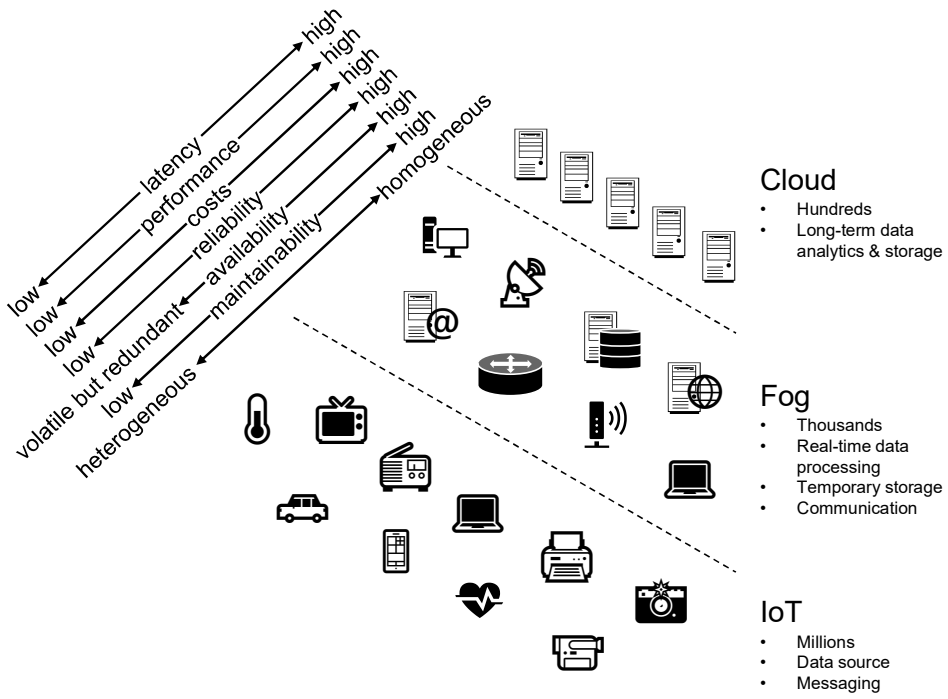


Figure 2.3 – Cloud vs. Fog vs. IoT

Comparison of Cloud, Fog and the Internet of Things (IoT). The lines between the areas are not continuous as some devices can switch back and forth between them. This figure is a mashup of (54) fig. 5, 6 and (42) fig. 1.

Why NFN?

Knowing the principles of cloud and fog computing, it now gets clear why NFN matches the concept. Due to the lower (single-device) performance in the fog and the IoT, processing must happen external from data sources and

in parallel. Parallelism is also required by the geographical distribution of involved devices and infrastructure. NFN with its ability to spilt and distribute computations, is a capable candidate to master this task. Furthermore, the interplay of NFN and CCN/NDN's pervasive caching, allows to pull vast amounts of (sensor) data from IoT devices and efficiently accumulate them, before they congest the network. Timely processing and fast reactions require task offloading from the cloud, too. The fog is the place where IoT and cloud meet, NFN the tool to realize it.

However, we so far deliberately neglected to mention that NFN has two levels of computations, λ-calculus on the top level for the orchestration of computations, and native code execution on the second level for the execution of computations. Unfortunately, native code execution can endanger the regular or expected behavior of NFN. The next section delivers further background knowledge on these problems.

2.5 The Risks of Referential Opacity

NFN and CCN/NDN each put forth one way to improve efficiency. On the one hand, running computations in parallel can be more time efficient than linear execution in many scenarios. Through the decentralized distribution of computations by NFN, truly parallel execution seems evident. On the other hand, CCN/NDN raises the hope that many computations do not even have to be executed as their results may be found in caches. However, the usage or interconnection of NFN with programming languages that support *referentially opaque* functions torpedoes both ambitions.

Referential Opacity vs. Referential Transparency

Referential opacity (RO) is the contrary of *referential transparency* (RT). A *routine* is either a *function* that is defined by having a return value or an *action* that is defined by not having a return value. The decision if a routine is either RO or RT depends on two properties, namely if the routine has side effects or not (see 2.5.1) and if the routine is deterministic or not (see 2.5.2). If a routine has *either* side effects *or* is non-deterministic, the function is said to be *referentially opaque*, or in short just opaque.

side effects ∨ non-deterministic ⇔ referentially opaque

If the function does *neither* have side effects *nor* is non-deterministic, the function is said to be *referentially transparent*, or in short just transparent.

¬side effects ∧ (¬non-)deterministic ⇔ referentially transparent

Note that actions are in general referentially opaque. If a process has no return value and no side effect (and therefore is referentially transparent), it has no effect at all. The remainder of this section will concentrate on details of side effects and determinism, the two fundamental properties of RT/RO. Moreover, it will be outlined that these properties are critical when it comes to multithreaded, parallel and distributed computations. In focus are related problems and especially how they are addressed. The ways how NFN and

CCN/NDN address the problems, or rather how they miss to address them, follows in the subsequent chapter 3.

2.5.1 Side Effects / Purity

Routines are said to have *side effects*, being *side effecting* or *impure* if they modify some state *outside* of their scope. Similarly, routines without side effects are called *pure*. In presence of side effects, the *order of evaluation* is crucial. Two brief examples[17] demonstrate this. Code Listing 2.1 is straightforward while Code Listing 2.2 shows how important it is to precisely understand a language reference before even using trivial instructions.

Code Listing 2.1 – Side Effect on a Static Variable

The functions KplusTwo() and KdivTwo() both change the static variable k that is outside of their scope. Therefore, it matters in which order the two functions are called, e.g. when they are used as input for a third function DoSomething(x,y).

```
1  // variable declaration
2  static int k = 10;
3
4  // function declaration
5  int KplusTwo(){ k = k + 2; return k; }
6
7  // function declaration
8  int KdivTwo(){ k = k / 2; return k; }
9
10 // action declaration
11 void DoSomething(int p1,int p2){ … }
12
13 // action + function calls
14 DoSomething(KplusTwo(),KdivTwo());
```

If the arguments of routine call `DoSomething` (Code Listing 2.1, line 14) are evaluated sequentially, the user actually calls `DoSomething(12,6)`. If the arguments are evaluated in reverse order, the user calls `DoSomething(7,5)`.

Widely-used languages like Java, C++ and C# know the increment operator (++) as syntactic sugar for the assignment command `x = x + 1`. All three languages also allow the compound assignment operator (+=), resulting in `x += 1`. Less known is that in all these languages, the assignment command has a return value, no matter which notation is used ((55), (56), (57)). The returned value when using the single assignment (=) or compound assignment (+=) operator is the incremented value, i.e. the value after the addition. The return value when using the increment operator (++) depends on the position of the operator. If used *prefix*, the incremented value is return, if used *postfix*, the original value is returned. Thus, the rather strangely appearing syntax of Code Listing 2.2, line 7 is perfectly valid C# syntax.

[17] The syntax is freely chosen but close to C# and Java.

Code Listing 2.2 – Side Effects Caused by Assignment Commands

The purpose of the assignment command is nothing else than changing the state of a variable. Thus, the assignment command is a side effecting routine. Because the routine does not have an identifier/name, it is called anonymous. If the assignment command is implemented to additionally return a value, the operation can be considered as anonymous (nameless) function.

```
1  // action call 1
2  int j = 0;
3  DoSomething(j++,++j);
4
5  // action call 2
6  j = 0;
7  DoSomething(-1 + (j = j + 1),j = j + 1);
8
9  // action call 3
10 j = 0;
11 DoSomething(PostIncr(ref j),PreIncr(ref j));
12
13 // function declaration
14 int PostIncr(ref int x){
15    try { return x; } finally { x = x + 1; }
16 }
17
18 // function declaration
19 int PreIncr(ref int x){
20    x = x + 1; return x;
21 }
```

Watched from a different angle, increment, decrement and assignment operators are *anonymous (nameless) functions*, returning a value, *and* having the side effect of changing the value of a variable, outside of their scope. Thus, the three action calls in lines 3, 7 and 11 of Code Listing 2.2 are equivalent. Given the evaluation ordering is heeded, the calls are always `DoSomething(0,2)` and `j` always equals to 2 directly after the call. If any of these calls would evaluate its arguments from the right to the left, the call would be `DoSomething(1,1)` and `j` equals to 2 directly after the call.

For this reason, language specifications normally define a universal rule about the order of evaluation, e.g. that routine arguments are evaluated from left to right. Java and C# implement the given rule (58), (59). However, C and C++ language specifications explicitly do not regulate the order of evaluation in any case (60), (61). Compilers may enforce individual rules. Consequences may be unexpected, non-deterministic results and exceptions that are almost impossible to trace back. Having a clear notion of ordering and its effect on results is even more important as soon as code is evaluated concurrently. The most common form of concurrency is multithreading. Basic concepts are introduced below, follow by a discussion why the solution statements may be difficult to apply on NFN.

2.5.1.1 Approaches in Parallel Systems

Imperative Programming

To lead from a historic perspective to the same problem, qualities of the early but still omnipresent *imperative* programming paradigm can be studied. It is characterized by a *sequential execution* of statements in the program code. Hence,

an imperative program code is a step-by-step description of what must be computed. Through this linear attention handling of instructions, a clear definition of 'before' and 'after' exists. This determinism has the invaluable advantage that *race conditions* and *data incoherence* are impossible. Beyond that, the paradigm perfectly fits the von Neumann architecture with its single instruction / single data structure. The strictly chronological program execution further entails that 1. the *current program state* is *known*, *unique* and *available* and 2. every instruction is executed exactly when specified and as often as specified. Furthermore, this way of execution allows interaction between program and outside world in form of program input and program output (I/O). If there is only one program state and only one party that can manipulate this state at the same time, unexpected states are excluded. Program execution according to expectations obviously was and still is a crucial qualification.

Multi-threading

However, this in turn implies that the program execution halts whenever a program input is required and waits until the input is available. It furthermore implies that only one process can work on the same data, i.e. reading or writing it. With the advent of multicore CPUs and the accompanying desire for accelerated program execution, new approaches were needed. *Processes* usually have their own state information and are independent of each other. This makes it easy to run multiple processes in parallel on a multicore CPU. In contrast, *threads* are sequential flows of work within a process. They share the same process state. Multithreading is also possible on a single core CPU. CPU time is allocated alternating to threads. This pseudo-parallelism may accelerate an overall process, e.g. when one thread anyway must wait on another event. This concept enables beneficial options like responsive processes that continue execution while waiting/listening to user input. Indeed, true *concurrency* of a process, i.e. parallel execution of several threads at the very same time, is only possible with multithreading on a multicore CPU.

Mutex & Semaphore

Nevertheless, the access to a shared state must be coordinated in both cases, not only in the sense that reading and writing collisions are omitted, but also in the sense that the program logic remains. To achieve the former, a mechanism called mutual exclusion (mutex) is used. A *mutex* is a lock associated with a state. The lock coordinates that only one thread or process has access to the state at a time. Metaphorically speaking, only one thread or task can possess the key to the mutex lock at a time. However, a mutex does not necessarily preserve the order of executions. If order matters, a concept called *semaphore* can be used. In essence, it is a signaling mechanism that instructs certain threads to *wait* on completion of other threads, respectively that instructs completed threads to *notify* waiting threads. However, the challenging task of correctly organizing signals is left to programmers. Forgotten notifications can frustrate successful program completion.

Initially, a shared state is not available for other cores or remote computing nodes. Hence, transferring only computing instructions is not enough. There are two options to handle the situation. Either a *copy of the state* can be shared along with the statements, or the node must always be in touch with *a unique state* whereof the access is shared. Both options have their drawbacks.

Cache Coherence

Copied states do not harmonize well with a mutex. Locking different copies independently of each other cannot guarantee the avoidance of the readers-writer problem. For example, multicore systems with individual caches for each core suffer from this problem. Therefore, so-called *cache coherence protocols* are

consulted. Basically, what they do is to tag individual cache lines with certain markers in each cache. For example, the MSI protocol uses the markers '*m*odified', '*s*hared' and '*i*nvalid'. All cache lines that are marked 'shared' are up-to-date and can be used. If a certain cache line is 'modified' in one cache, it is marked as such, while the equivalent cache lines are marked 'invalid' in all other caches. They do no longer hold the up-to-date state. Before they can be used again, they must be updated with the up-to-date information from the 'modified' cache. The protocol promises to keep caches coherent. Simultaneously, it omits unnecessary cache refreshes that would entail avoidable latency and transmissions between the caches. The protocol was further optimized through the extensions MESI with an extra '*e*xclusive' status and MOESI with an additional '*o*wned' status.

A unique state can implement a mutex without a cache coherence protocol. However, allocating the key of the mutex to multiple requesters requires a queue. Generally, queues are prone to information loss, caused by dropped packets. Making sure that all updates were applied requires communication between requesters and the node with the unique state. While this communication overhead may be tolerable within a CPU, i.e. between the cores, it may slow down a computation noticeably within a network, depending on RTTs.

The same applies to semaphore signaling. Communication takes place between two requesters who not necessarily know each other. Hence, an intermediary that knows all involved nodes is again needed. Although possible, the intermediary must not necessarily be identical with the node holding the shared state.

Atomic Counter

However, putting up some restrictions makes state sharing less cumbersome. For example, Amazon's ElastiCache offers so-called *atomic counters* (62). An atomic counter is a unique (non-replicated) numerical value that only can be incremented, for instance to count the number of visitors of a website. As increments are associative, a property that will be discussed below in more details, they can be applied in the order they arrive. Therefore, no coordination to establish order is needed. A mutex on the counter is enough to avoid write collisions. However, slight over- or undercounting is possible as increments may fail unnoticed or mistakenly applied multiple times. Nonetheless, many scenarios are not qualified for making such compromises. For example, a bank account service must maintain the order of transactions to avoid negative balances. Transactions must also follow an 'exactly-once' semantic in order not to artificially create or erase money.

2.5.1.2 CRDT

Although not replicated, the atomic counter implicitly fulfils some requirements to be categorized into a class of exceptional data types called *Conflict Free Replicated Data Types* (CRDT). According to Shapiro et al. (63), their main feature is that *strong eventual consistency* (SEC) among replicas can be preserved without coordination. In other words, CRDT allows to independently work on multiple *copies* of a state, guaranteeing that the copies will converge, i.e. being in a consistent state at some point in time.

CmRDT

Despite this attractive feature, there are quite some restrictions that limits the application spectrum of CRDTs. *Operation-based* CRDTs (a.k.a. Commutative Replicated Data Types (CmRDT)) only work with *update operations* that are *commutative*. Great attention must be paid to the following differentiation: The commutativity property is commonly defined on binary operations. It says that the two operands/arguments 'x', 'y' of the binary operation 'op' can be interchanged without changing the result. With other words, the *order of operands* is insignificant.

Commutativity $op(x,y) = op(y,x)$

For example, the addition "+" operation on $\mathbb{R}$ is commutative because $x+y = y+x$, e.g. $2+5 = 5+2$. In contrast, subtraction "-" and division ":" operations are not commutative on $\mathbb{R}$ because $x-y \neq y-x$ and $x:y \neq y:x$, e.g. $2-5 \neq 5-2$ and $2:5 \neq 5:2$. However, what matters for CmRDT is that the *order of operations* is insignificant. For example, the unary *increment* (i.e. add1) and *decrement* (i.e. subtract1) *operations*, as closed set of allowed operations, commute because decrement(increment(x)) = increment(decrement(x)), i.e. $(x+1)-1 = (x-1)+1$.

Idempotence of the operation, i.e. the requirement that an operation has the same effect to the result if applied once or multiple times, is *not* a mandatory requirement. However, making sure that non-idempotent operations are applied only once per copy is obviously inevitable. This task must be ensured by the system. Idempotence has the following general definition:

Idempotence $op(op(x)) = op(x)$

CvRDT

Alternatively to CmRDTs, *state-based* CRDTs (a.k.a. Convergent Replicated Data Types (CvRDT)) can be used. CvRDT do not exchange operations to be applied on the state. Instead, full states are exchanged. A node receiving an updated state must merge its own state with it. The *merging function* requires to be commutative (like the update function in CmRDT) but additionally *idempotent* and *associative*, too. Associativity says that if an operation occurs more than once, the *order of operations* does not change the result.

Associativity $op(x,op(y,z)) = op(op(x,y),z)$

For example, the addition "+" operation on $\mathbb{R}$ is also associative because $x+(y+z) = (x+y)+z$, e.g. $2+(5+6) = (2+5)+6$. Again, subtraction "-" and division ":" operations are not associative on $\mathbb{R}$ because $x-(y-z) \neq (x-y)-z$ and $x:(y:z) \neq (x:y):z$, e.g. $20-(5-4) \neq (20-5)-4$ and $16:(4:4) \neq (16:4):4$. Applied to a scenario where one node with its own state receives two updated states, the associativity property allows to merge them in arbitrary order.

Partially Ordered Sets

A short mathematical detour helps to understand how this fits together with the problem of state updates. A set and the two binary operations infimum[18] and supremum[19] (both are associative, commutative, idempotent and connected by the absorption law) define a *lattice*, i.e. a *partially ordered set* with a unique supremum and a unique infimum for every two elements of the set.

[18] a.k.a. greatest lower bound or meet
[19] a.k.a. least upper bound or join

Absorption Law op1(x,op2(x,y)) = op2(x,op1(x,y))

This axiom can be used to decide which state is newer for any two states, independent of the order in which they arrive and being merged. The only rule that must be observed is that *update functions* must be *monotonically non-decreasing* according to the lattice, i.e. they must not decrease the internal state. Decreasing the internal state would mean to go backwards in the partial order, i.e. reverting to *older* states. However, "updates" that do not change the state are not of interest. Therefore, the examination can be confined to *monotonically increasing* update functions, i.e. updates that create *newer* states. For the reason of completeness, it is also demonstrated that the two operations infimum and supremum conform to the laws and therefore actually define a lattice. For this example, we assume x > y > z.

Commutativity

$$\begin{aligned} & \inf(x,y) = \inf(y,x) \\ \Leftrightarrow\ & y = y \end{aligned}$$

$$\begin{aligned} & \sup(x,y) = \sup(y,x) \\ \Leftrightarrow\ & x = x \end{aligned}$$

Associativity

$$\begin{aligned} & \inf(x,\inf(y,z)) = \inf(\inf(x,y),z) \\ \Leftrightarrow\ & \inf(x,z) = \inf(y,z) \\ \Leftrightarrow\ & z = z \end{aligned}$$

$$\begin{aligned} & \sup(x,\sup(y,z)) = \sup(\sup(x,y),z) \\ \Leftrightarrow\ & \sup(x,y) = \sup(x,z) \\ \Leftrightarrow\ & x = x \end{aligned}$$

Idempotence

$$\begin{aligned} & \inf(\inf(x)) = \inf(x) \\ \Leftrightarrow\ & x = x \end{aligned}$$

$$\begin{aligned} & \sup(\sup(x)) = \sup(x) \\ \Leftrightarrow\ & x = x \end{aligned}$$

Absorption Law

$$\begin{aligned} & \inf(x,\sup(x,y)) = \sup(x,\inf(x,y)) \\ \Leftrightarrow\ & \inf(x,x) = \sup(x,y) \\ \Leftrightarrow\ & x = x \end{aligned}$$

Grow-Only Counters & Sets

Due to these demanding restrictions, the building blocks of CRDTs are rather simple data structures such as counters or sets. State-based CRDTs (CvRDTs) are often called *grow-only* data structures due to the restriction that their states can only be increased monotonically. For this reason, replicated counters are often referred to as grow-only counters, or simply G-counters, when implemented as CvRDT. A more sophisticated positive-negative (PN-)counter can be created by the composition of two G-counters. Although both counters can only grow, the implicit counter state is given by subtracting the N-counter from the P-counter. The same is possible with *sets*. Items can only be added to a grow-only (G-)set. A two-phase (2P-)set consists of two G-sets, the addition set holding all ever added items, the remove set holding all ever removed items.

The implicit set state is again given by ignoring all elements from the remove set in the addition set.

2.5.1.3 The 'Concealing' Trick

Yet to mention is a trick to conceal side effects. In fact, every *impure* routine can be remodeled to a *pure* routine. If a routine should not manipulate state outside of its scope, the whole affected state can explicitly be copied inside the routine, be manipulated therein, and finally the updated/manipulated copy of the state can explicitly be returned along with other possible return values. As an example, imagine a function that returns the cubic number of the input and increments a counter in the environment (lines 1-8 in Code Listing 2.3). This impure function can be remodeled to a pure function by passing the affected counter as argument and additionally return the changed updated counter along with the results, e.g. as a tuple type (lines 10-17 in Code Listing 2.3).

Code Listing 2.3 – The 'Concealing' Trick

Impure routines can be remodeled into pure routines by explicitly copying the affected state into the routine and explicitly returning the modified copy of the state.

```
 1 // variable declaration 1
 2 static int counter = 15;
 3
 4 // impure function declaration
 5 int ImpureCubic(int arg0){
 6    counter++;
 7    return arg0 * arg0 * arg0;
 8 }
 9
10 // variable declaration 2: static no longer needed
11 int counter = 15;
12
13 // pure function declaration, returning a tuple
14 (int res, int newCounter) PureCubic(int arg0,
   int oldCounter){
15    int newCounter = oldCounter + 1;
16    return (arg0 * arg0 * arg0, newCounter);
17 }
```

In fact, the responsibility for handling the side effect is shifted from one layer (within the function) to another layer (outside the function). The environment of the function must now decide what to do with `oldCounter` and `newCounter`, e.g. overwriting the old state with the new state (which is the process of handling the side effect) or keeping both states. This responsibility handover can be consecutively repeated to the outermost layer.

Ostensibly, the order of evaluation does now no longer matter to preserve the program's logic. What now matters is where `oldCounter` is still used and where `newCounter` is used. Eventually, this trick enforces an implicit ordering of events. `newCounter` cannot be used before it is available. This narrows the possibilities for true parallelism. However, this is not a drawback of this particular solution, but a very general and inevitable problem of side effects. Nevertheless, the downside of this trick is obvious. Imagine that `PureCubic` is called 100 times. There are now 100 copies of the counter. For each of them,

one must decide if the copy should be kept for a later reuse, or if the copy can be discarded. For those reasons, it is rather unusual to apply the 'concealing' trick. For example, the I/O *monad* in the Haskell programming language bases on this concept. Other languages, e.g. Clean, use the trick in combination with a *uniqueness type* to guarantee referential transparency in combination with referentially opaque functions. A state used as argument for a parameter that has been declare 'unique' cannot be referred ever again. The user is forced to use the updated copy of the state.

2.5.2 Determinism

Making matters worse, violating the order of evaluation of impure functions is not the only possibility to generate non-deterministic results and behavior. As mentioned in 2.2.1 and the introduction of 2.5, CCN/NDN raises the hope that many computations do not even have to be executed as their results may be found in some caches. However, computations can only profit from cached results if these results are (still) valid. Unfortunately, results can lose their validity over time, i.e. getting *stale*, on location changes, on varying workload and many other *external factors*. And even worse, it is commonplace in code-writing to use functions that return such changing results. An example of a function that produces results which lose validity over time is

```
string DateTimeNow()
```

that returns current date and time, formatted as a string. Given that a long enough interval elapses between the calls, this function returns a different result upon every new call. Moreover, results may depend on the time zone setting of the machine that executes the call. Results of the call

```
int ProcessorCount()
```

that returns the number of available cores depend on the (virtual) machine that executes the call. Further examples with their dependence are found easily:

```
Foo GetGPSCoordinates()   // on geographical location
long GetAvailableRAM()    // on load
string ReadLine()         // on user input
```

Determinism

All these examples need not to be side effecting. Nevertheless, functions with such dependencies on external factors are called *non-deterministic*. If they are independent from outer conditions, they are called *deterministic*. Accordingly, deterministic functions *always* return the exact same value for a specific set of arguments. Determinism is desirable functionality because it makes code transportable without corrupting its logic. No matter by whom or where the call is executed, the result will be the same. To give a positive example, too, math is deterministic:

$5+2 = 7$ (always)

$11/3 = 3.\overline{6}$ (always)

$\sin(3/4\pi) = 1/\sqrt{2}$ (always)

2.5.2.1 Approaches in Parallel Systems

Caching

Rolled-up again from a computational and efficiency-driven perspective, improving the execution time became the focus, especially because memory got cheap and available in vast extents. Thus, it was self-evident to trade time complexity against memory complexity. One approach, assuming that lookup is faster than re-computing, is to cache previously computed results in an *associative array* where it can be retrieved upon new requests for the same computation. Associative arrays are unordered key-value lookup tables (LUT), often implemented as hash tables. Hence, if hashing the name, a rather cheap operation, has a lower time complexity than re-computing the result, caching improves time complexity at the expense of a LUT entry, implying an increased memory complexity. The potential benefit can be considerable whenever a function has high time complexity.

However, not all types of results are equally suitable for caching. If the side effect matters, results from impure functions cannot be cached. Results from deterministic functions seem more suitable to be cached because they cannot get stale. Non-deterministic results can be cached, too. However, this requires a carefully worked out *cache control mechanism* such that users are not served with stale results.

Memoization

Obviously, the optimal case is when the function is pure and deterministic, i.e. when being referentially transparent. In this case, the result can be cached and satisfy later request without having to worry about staleness or side effects. The result can literally be cached forever, given that enough memory is available. In computing, the strategy to cache and make use of referential transparent results is called *memoization*. Both caching and accessing results must explicitly be implemented by the software engineer. If done wrongly, e.g. by caching and accessing opaque results, the intended purpose of the software may be missed. The classic example is recursive computation of Fibonacci numbers: `fib(n) = fib(n-1) + fib(n-2)`. If `fib(n-1)` and `fib(n-2)` are known and available, `fib(n)` can easily be computed. If the two summands are not available, it is rather costly to compute them if `n` is large. Once known, the individual summands can be cached in an associative array. Note that recursion is no precondition for memoization. Every RT result can be cached and reused later. The example further clarifies that memoization is rather a run-time than a compile-time optimization. An associative array gets filled consecutively whenever it was needed to compute a result. It does not make sense to compute results preemptively, not knowing if they will ever be used. Memoization is a logical concept rather than a machine-dependent optimization. Together with its run-time aspect, it is suitable to be applied across platforms. Its deployment only requires an associative array. Fortunately, CCN/NDN provides such an associative array pervasively: the content store.

Distributed Caching vs. Memoization

However, note that the motivation for memoization and distributed caching was originally not the same. Memoization mainly aims at accelerated computations while distributed caching aims at an overall increased throughput of a network. The main purpose of *Content Delivery Networks* (CDN) is to curtail data delivery to smaller regions and therefore to prevent other regions of the Internet from data delivery duties. In turn, these less congested regions should have more available capacities to satisfy other requests, increasing the overall throughput of the Internet. The latency aspect of caching in order to accelerate computations became important not until the spread of the IoT and the

accompanying needs of edge and fog computing. Examples like increased responsiveness in gaming or real-time reactions on sensor data have already been discussed in section 2.4.

3. The Soft Spots

As outlined above, the integration of native code execution as done by NFN requires to treat referential opacity correctly. This chapter starts by showing where NFN is not precise enough in this matter and by identifying drawbacks arising from certain design considerations. In the second section, we explain why it is required to be clear about the specific interpretation of expressions in order to obtain logically referentially transparent results. The third and last section shows that referentially transparent computations alone are not enough to ensure a deterministic resolution system. We will see that CCN and particularly NDN allow non-deterministic requests.

Chapter	Section
Background	3.1 Tensions Between NFN and Referential Opacity
The Soft Spots	3.2 Computability Limitations
Equivalence Classes	3.3 Non-Deterministic NDN
Protocol & Architecture Adaptions	
Evaluation	
Related Work	

TL;DR – Key Messages

- ✓ NFN's λ-expressions are referentially transparent. However, NFN handles invoked binary functions always the same, no matter if they are referentially transparent or referentially opaque. To preserve functional correctness, i.e. avoiding wrong results, propagated advantages such as arbitrary order of evaluation, reuse of cached results, and concurrency must be waived.
- ✓ Pure mathematical functions that are referentially transparent may yield different results when computed on a physical machine. This has two reasons: 1. limited floating-point accuracy and 2. some solutions to mathematical functions cannot be represented as a finite sequence of basic arithmetic operations (on integers). They can only be approximated.
- ✓ A name-based content retrieval system is deterministic if: 1. Content objects are immutable. 2. Once a name was used, it must not be reused for a different content. 3. The name resolution mechanism must not allow non-deterministic results.

✓ NDN's name resolution mechanism is not deterministic. To blame for this are the 'CanBePrefix' selector and the implicit digest component that is not mandatory part of interests (in explicit form).

3.1 Tensions Between NFN and Referential Opacity

According to Sifalakis et al. (38), binary data manipulation is inefficient if solved on name-manipulation level, e.g. as done by the λ-calculus. For example, there is no direct notion of the addition, making it quite hard to express '+3'. According to Blaheta (64), a strategy is to first define a successor function (`succ`, i.e. add1) which is then reused in the definition of an `add` function. Adding 'a' to 'b' is equivalent to apply `succ` 'b' times to 'a'.

```
succ ≡ (λ(n)(λ(f)(λ(x)(f((n f)x)))))
add ≡ (λ(a)(λ(b)((a succ)b))
```

Worse yet, even natural numbers must be defined first. Commonly, this is done with *Church numerals*. '3' could be expressed like (also (64)):

```
3 ≡ (λ(f)(λ(x)(f(f(fx)))))
```

Task Split

Multiplication, factorial and other mathematical concepts are then increasingly elaborate. To avoid this, users are encouraged to resort to more expressive programming languages for the definition of functions like `succ` and `add`. This *second level of computations* is where the actual execution of functions is taking place, i.e. finding values that replace λ-expressions. NFN's reference implementation (65) uses Scala that generates Java byte code (38). These functions must be compiled and deployed in the network before they can be used. Therefore, NFN's λ-style *first level of computations* degenerates to mere lookup of named data and opportunistic distribution of computations.

Rather Eager

A first advantage, that almost completely falls victim to this two-layered computation approach, is the delaying call-by-name evaluation strategy. This non-strict evaluation strategy will appear rarely in reality. Due to the extremely constrained readability of λ-expressions, almost every single operation will exist in the form of a native function. As Sifalakis et al. described in (38), a call to a native function will immediately switch the evaluation strategy from (non-strict) *call-by-name* to (eager/strict) *call-by-value*. This means that every argument expression needs to be entirely evaluated before the actual call can take place. For the same reason, one must assume that these argument expressions are again calls to native functions.

Rather Wasteful

Environment, argument stack and result stack from computation configurations are all states and they change frequently. A direct consequence of the immutability property of those objects/states is that even simple expressions induce an avalanche of copied states. Most of them become superfluous shortly after they have been created. Once that an (intermediate) result is available, all states that led to this result will never be accessed again. Although copies may eventually vanish from caches, garbage collection is needed to confine memory waste. This is an acknowledged drawback (3). In

order to improve garbage collection, it would be helpful to work with mutable states and to know which copies can be deleted preferably because they will never be used again.

Rather Sequential

Getting worse, the promoted parallel processing sequence can strongly be questioned because NFN is unclear about referential transparency. Undoubtedly, λ-calculus at the top level of NFN is RT. However, the differentiation if the whole system is RT though, depends on the second level of computation. Calls to referentially opaque routines nested in λ-expressions cause the whole expression to be opaque. In (66), authors outline the limiting aspect of NFN's purely functional approach, arguing that all tasks requiring side effects cannot be covered. This statement clearly implies that the second level of computation is restricted to be RT. In contrast, the statement that result stacks are capable of "holding function side effects during λ-expression evaluation" (3) is a very clear hint at that native functions may also be RO. Somehow or other, there must either be a restriction on allowed programming concepts for the implementation of native routines or a control mechanism that differs RT routines from RO routines. In this thesis, we will investigate on the latter approach. If this is not done, expressions must either be evaluated sequentially in combination with the concealing trick, or wrong results can appear, mainly through the unintended reuse of cached results and incorrect order of evaluation. For example, imagine having an arbitrary `/OuterFunction` and a `/Random` function that returns random values. It matters if `/Random` is evaluated once (before the substitution as in $\lambda_{3.1}$) or twice (after the substitution as in $\lambda_{3.2}$), as well as if the result is cached (and reused) or not.

```
λ3.1:  (λx.(/OuterFunction x x)) /Random
λ3.2:  /OuterFunction /Random /Random
```

3.2 Computability Limitations

Not bad enough yet, pure mathematical functions that are referentially transparent may yield different results when computed on a physical machine. This has two reasons: 1. limited floating-point accuracy and 2. some solutions to mathematical functions cannot be represented as finite sequence of basic arithmetic operations (on integers). They can only be approximated. One must then decide how far the true solution should be approximated. Often, [1.] aggravates the problem derived from [2.].

1) Accuracy

Three short examples are given to study the relationship of referential transparency and computational accuracy. To begin with, binary representation of 0.1 (= 1/10) is inaccurate. Table 3.1 shows four different binary representations of 0.1, yielding three different decimal values. All four configurations follow the specifications IEEE 754-1985 (67) and IEEE 754-2008 (68) standard for binary floating-point arithmetic.

Table 3.1 – Floating-Point Accuracy

This example shows the imprecise representation of 0.1 (=1/10) in binary floating-point form. Precision and rounding strategy can influence the outcome. Strictly interpreted, these representations should render every containing expression opaque because they deviate from the true value.

Configuration Number (CN)	Description: IEEE 754
1	single precision, not rounded
2	single precision, rounded[20]
3	double precision, not rounded
4	double precision, rounded[20]

CN	Binary
1	00111101 11001100 11001100 11001100
2	00111101 11001100 11001100 11001101
3	00111111 10111001 10011001 10011001 10011001 10011001 10011001 10011010
4	00111111 10111001 10011001 10011001 10011001 10011001 10011001 10011010

CN	Binary in Decimal
1	0.099 999 994 039 535 522 460 937 5
2	0.100 000 001 490 116 119 384 765 625
3	0.100 000 000 000 000 005 551 115 123 125 782 702 118 158 340 454 101 562 5
4	0.100 000 000 000 000 005 551 115 123 125 782 702 118 158 340 454 101 562 5

CN	Error
1	$-5.9604644775390625 \times 10^{-9}$
2	$1.490116119384765625 \times 10^{-9}$
3	$5.551115123125782702118158340454101562 5 \times 10^{-18}$
4	$5.551115123125782702118158340454101562 5 \times 10^{-18}$

CN	Accuracy [%]
1	99.999 994 039 535 522 460 937 5
2	99.999 998 509 883 880 615 234 375 0
3	99.999 999 999 999 994 448 884 876 874 217 297 881 841 659 545 898 437 5
4	99.999 999 999 999 994 448 884 876 874 217 297 881 841 659 545 898 437 5

None of them is an exact result. Hence, it is questionable if any of them should be considered referentially transparent. Referential transparency would allow, during reduction, to substitute the expression "1/10" with the computed value, *without* changing the outcome.

Moreover, the above example shows that rounding can cause both, slight *over- and undercounting*. The following example shows that over- and undercounting can alternate in an iterative process like e.g. (finite) summation.

[20] According to the rounding rule described in (68), section 4.3.1, p. 16: "Rounding-direction attributes to nearest" → "roundTiesToEven"

Table 3.2 – Repeated Rounding & Accuracy

To demonstrate the effects of repeated rounding and accuracy, the following test logic is implemented in four different programming languages:

The "sum", represented as single precision floating-point number, starts at 0.0. It is increased by 0.1 in a loop as long as the sum is below, i.e. less than, a certain guard. The number of completed loops is counted with a 64 bit integer.

Source codes can be found in appendix section 9.1, Code Listing 9.1 to Code Listing 9.4.

		Single Precision		
		#loops		
Language	Guard	expected	measured	delta
C99[21]	9'999.9	99'999	100'014	+15
	99'999.9	999'999	990'563	-9'436
	999'999'999.9	9'999'999'999	non-terminating	–
C# 7.3[22]	9'999.9	99'999	100'014	+15
	99'999.9	999'999	990'563	-9'436
	999'999'999.9	9'999'999'999	non-terminating	–
Java 8[23]	9'999.9	99'999	100'014	+15
	99'999.9	999'999	990'563	-9'436
	999'999'999.9	9'999'999'999	non-terminating	–
Python 2.7[24]	9'999.9	99'999	99'999	0
	99'999.9	999'999	999'999	0
	999'999'999.9	9'999'999'999	9'999'998'368	-1'631

Floating-point accuracy is relative to precision *and* involved numeric value. If the difference between two summands grows, the effect of rounding aggravates. Consequently, the overall accuracy decreases. 9'436 / 999'999 corresponds to an error ~0.9436%, yielding an accuracy of ~99.0563%. The effect can even get that worse that adding another 0.1 to the sum no longer changes the sum. In this particular example, the critical point is reached when the sum has grown to 2'097'152. At that moment, the exponent changed from 2^{20} to 2^{21}. The smallest possible change to a floating-point number is to increment the last bit of the mantissa. With an exponent of 2^{20}, such an increment yields a change of 0.125, with an exponent of 2^{21} a change of 0.25. As 0.1 is closer to 0 than to 0.25, the add operation has no effect. Therefore, the sum will never reach the guard and the program will never terminate.

Another interesting insight is that the loop is potentially left *before* the loop condition is 'false'. For example, the C99 program with a guard setting of 99'999.9 leaves the loop after 990'563 iterations. However, the sum grew only to 99'999.8984375 at that time. Thus, the loop condition should still evaluate to 'true' and one iteration more should be taken. However, this does not happen because 99'999.8984375 is closer to 99'999.9 (deviation = 0.0015625) than 99'999.90625 (deviation = 0.00625), which is the number that results when incrementing the least significant bit. More precisely, the problem is that

[21] Visual C++ Compiler Version 19.14.26433
[22] Visual C# Compiler Version 2.8.3.63029 (e9a3a6c0)
[23] javac 1.8.0_181
[24] Python 2 language interpreter Version 2.7.14 (Cygwin)

99'999.9 cannot be exactly represented. In fact, the guard is compared to 99'999.8984375 as this is the closes representation of the true value. Hence, it is obvious that the 'less than' condition no longer holds.

The above example includes and reveals the problem of adding small numbers to large numbers. Non-termination is only one phenomenon of effectless operations. Another phenomenon is that commutative and associative operations, e.g. addition and multiplication, may be non-associative when using floating-point arithmetic. Inspired by the example from Villa et al. (69), Figure 3.1 demonstrates how two different results may be obtained with just two additions. Instead of using decimal (base 10) numbers as Villa et al. (69) did in their example, we use 8 bit floating-point numbers (base 2) with 4 bit mantissa and 4 bit exponent. The mantissa is normalized to $1_{10} \leq m < 2_{10}$ according to the IEEE 754-2008 specification (68).

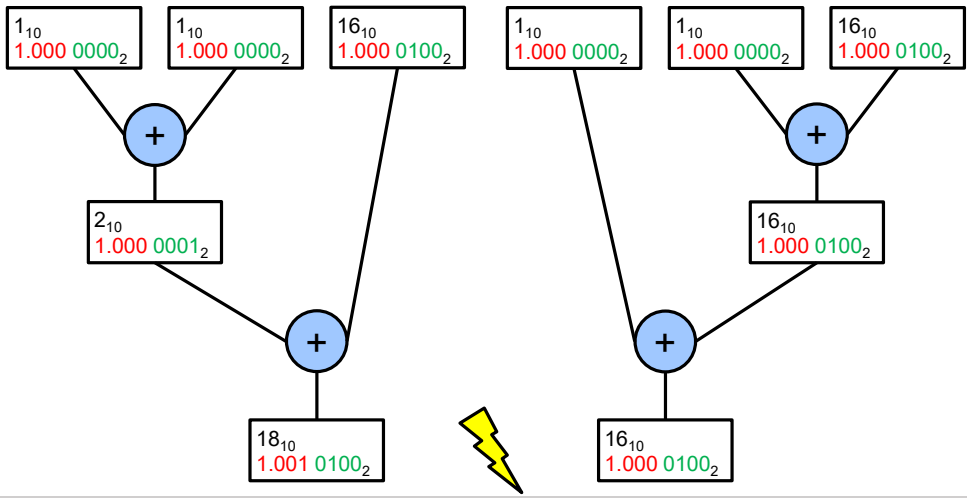


Figure 3.1 – Non-Associative Addition

When using floating-point arithmetic, even the order of commutative and associative operation, like e.g. the addition, matters. The reason is the limited precision of floating-point numbers. When adding two numbers, the exponent of the number with the smaller exponent is adjusted to the exponent of the number with the larger exponent. Doing so, the mantissa of the number with the (originally) smaller exponent is accordingly adjusted. If the mantissa does not offer enough bits, information may be lost, causing inaccurate results.

1_{10} = $1.000\ 0000_2 = (1*2^0)_{10}$ → $0.000(1)\ 0100_2$

2_{10} = $1.000\ 0001_2 = (1*2^1)_{10}$ → $0.001\ 0100_2$

16_{10} = $1.000\ 0100_2 = (1*2^4)_{10}$

Imagine the problem of array summation. One solution strategy is to add up iteratively all elements of the array. Another 'divide-and-conquer' solution strategy is to concurrently execute iterative summation on array segments with a subsequent summation of all subtotals. Comparing the approaches, the order of additions is not identical. Hence, the results are not necessarily the same.

Finally, the Python implementation is another proof for that different precisions can affect results. Python internally maps single precision floating-point numbers to double precision floating-point numbers. Thanks to the

higher precision, miscounted loop iterations and non-termination will appear "later", i.e. when the difference between two summands is even bigger.

2) "Difficult" Expressions

Iterative operations are also commonplace in math. Math differentiates classes of expressions with increasing "difficulties" for mathematical analysis. *Arithmetic* expressions comprise the four elementary arithmetic operations (+, -, *, /) on *integers*. Moreover, arithmetic expressions must be *finite*. *Algebraic* expressions extend the set of operations by raising to integer power and extraction of integer roots, i.e. rational exponents. Additionally, it allows the use of variables, not only integers. Algebraic solutions can be considered as zeros (a.k.a. roots) of n^{th} order polynomials. Still, expressions must be finite. Closed-form expressions are already considered "difficult" because they extend expressions by the "difficult" functions logarithm, trigonometric function, hyperbolic function and their inversions. These are so-called *transcendental* functions, meaning that they cannot be expressed as finite number of algebraic expressions. For example, the natural logarithm (ln) can only be expressed as *infinite* summation of algebraic expressions, or more precise as power series:

$$\ln(x) = 2 * \sum_{n=0}^{\infty} \frac{1}{2 * n + 1} \left(\frac{x-1}{x+1}\right)^{2*n+1}$$

However, these functions are "well-known", meaning that there are known transformation rules that ease the handling of them. Examples include:

$$\ln(x * y) = \ln(x) + \ln(y)$$

$$\tan(\varphi) = \frac{\sin(\varphi)}{\cos(\varphi)}$$

Analytic expressions cover convergent infinite sum, product and continued fraction, i.e. functions beyond the "well-known" ones. Finally, *mathematical* expressions extend the set of functions with limit, derivative and integrals.

As mentioned at the beginning of this section, the limited precision of floating-point numbers affects the computation of a mathematical approximation. It does not make sense trying to approximate a transcendental function beyond the precision of a floating-point number. Nevertheless, it is important to distinct where imprecisions derive from, i.e. from insufficient floating-point accuracy, potentially aggravated through repeated reuse of inaccurate results, or from insufficient approximation. For clarification of this distinction, Table 3.3 lists the iterative approximation to the transcendental number ln(7) = 1.9459101490553132, neglecting floating-point accuracy[25].

[25] All results according to http://www.wolframalpha.com/, using the input 2*sum((1/(2*n+1))*((7-1)/(7+1))^(2*n+1),n from 0 to 20)

Table 3.3 – Transcendental Functions

The natural logarithm is an example of a transcendental function. Almost all results are transcendental numbers and can only be approximated. Calling such functions do not necessarily return identical results. Therefore, they should be considered opaque. The result depends on a pre-set constant number of iterations, i.e. from an additional condition.

n (iterations)	Value / Result	Error	Accuracy [%]
0	1.5	0.44591	77.0848
1	1.78125	0.16466	91.5381
2	1.87617	0.0697383	96.4162
3	1.91431	0.0316	98.3761
4	1.931	0.0149145	99.2335
5	1.93867	0.00723542	99.6282
10	1.94567	0.000242717	99.9875
20	1.94591	4.27718×10^{-7}	99.999978

However, in terms of computability, difficulties begin before transcendental functions. Already finite algebraic expressions lead to problems because the extraction of integer roots can only be solved with an iterative approximation. Note that divisions often involve floating-point arithmetic, too. Nevertheless, the operation is usually not considered computationally "difficult" because it does not involve an iterative approximation process. Note that infinite sequences like the approximation of the logarithm can be expressed as λ-expressions. However, their reduction is non-terminating.

Now that we have seen that even referentially transparent expressions do not necessarily produce referentially transparent results on a computer, great attention must be paid when doing computations on names. Attractive features of functional approaches like arbitrary order of evaluation and "lazy evaluation"[26] do not necessarily lead to consistent results in hardware. Consequently, being clear about the specific interpretation of expressions is crucial to obtain logically referentially transparent results. The last section of this chapter outlines that producing referentially transparent results is not yet enough for a whole system to be RT.

3.3 Non-Deterministic NDN

NDN cannot have side effects as it is only about mere content retrieval and not about calling routines. Therefore, purity is not a problem. Determinism, however, should be considered carefully. There are three conditions for a system like NDN to be deterministic, and therefore referentially transparent. All of them must apply.

[A] Immutable content objects
[B] Once a name was used, it must not be reused for a different content.
[C] The name resolution mechanism must not allow non-deterministic results.

[26] The term "lazy evaluation" is normally associated with call-by-need evaluation that is the memoized variant of call-by-name. Nevertheless, call-by-name also delays the evaluation of arguments until they are needed. Therefore, the strategy can be called "lazy", too.

[A] and [B] enable to cache content objects everywhere and persistently. While [A] clearly applies to CCN/NDN, [B] also applies but causes problems concerning deterministic name resolution [C]. However, [C] does anyway not apply, particularly not to NDN, and independently of [B].

First, condition [C] will be investigated. Here, the two critical attributes are 'MustBeFresh' (70) and 'CanBePrefix' (71). Both can appear in NDN interests. While the former does not necessarily violate determinism, the latter does. Subsections 3.3.1 and 3.3.2 enlarge upon both attributes.

Second, condition [B] will be investigated. Both CCN and NDN make names unique by appending a digest component to every data packet. While this makes every two different data packets discriminable, it aggravates deterministic name resolution because this component is not necessarily part of content matching. Subsection 3.3.3 delves into the subject.

3.3.1 Freshness

Attributing a 'FreshnessPeriod' (72) to an *immutable* content seems like a contradiction at first. Expired freshness somehow suggests that the immutable content is no longer valid and therefore should no longer be used or supplied. However, immutable content objects are valid eternally. Nevertheless, they can get stale in terms of that newer information is available. For example, the World Wide Web is full of such evolving content. Because NDN content objects are immutable, there must be another way to ask for *latest* content. NDN solved this with the binary interest attribute 'MustBeFresh'. If the attribute is set to true, the freshness period, a non-negative integer, of the according content object must be greater than 0. If the freshness period is 0 or below, a.k.a. non-fresh, the content, although valid, must not be supplied. If the attribute is set to false or omitted, matching content can be delivered, even if the freshness period is 0 or below.

Solely not delivering non-fresh content does not satisfy the requester's desire for latest content. It just avoids unnecessary data delivery. The problem is how a requester should guess the right name of newer content. Often, immutable but evolving content is implemented with a versioning system. Attributing distinct version information to new content allows to disambiguate them. Version information does not have to be a growing number or a timestamp. Every mechanism that ensures distinctiveness to avoid collisions is suitable, including NDN's implicit digest name component (73) that is a mandatory (but implicit) part of every data packet's name. However, timestamps or other growing numbers have the advantage to offer a clear notion of 'newer', i.e. the larger the newer (74). Additionally, they are not affected from collisions. Colliding version information causes non-deterministic behavior.

The NDN protocol does not specify a certain approach how to get hold of version information of latest content. However, Shi (75) discusses possible solutions, including:

- Before asking for content, a requester can explicitly ask for version information in a separate round of data exchange. A random name component that is appended to the interest makes sure that no caches are hit, and that producers dynamically create new answers.

- If working with NACKs, this information can be included. Nevertheless, this involves two rounds of data exchange, too.

In both cases, the producer should offer the needed functionalities. Another possibility to retrieve latest, or at least fresh content is to introduce the 'CanBePrefix' attribute that influences the way content is matched.

3.3.2 Content Matching

The way how the name from an interest is matched against the names in data packets is crucial because it determines what a user receives upon its request. In the above discussion on freshness, we tacitly presumed that the name from an interest matches *exactly* the name in the data packet. Thus, if the version information (that is part of the name) from the interest does not exactly match the version information in the data packet, no data will be returned, independently of freshness. In fact, this is the default behavior of NDN version 0.3 if the binary 'CanBePrefix' attribute is set to 'false' or omitted. If the attribute is set to 'true', every data packet with a name that starts with the name from the interest can satisfy the request. It does not matter how many additional name components the name in the data packet has.

This is a novelty compared to NDN version 0.2. Back then, interests featured selector fields to specify how many additional name components *at least* ('MinSuffixComponents') and *at most* ('MaxSuffixComponents') a name in a data packet must have to match the interest. Figure 3.2 lists four different selector settings A, B, C, D for the interest `/newspaper/frontpage` and shows which data packets are matched.

Back to the freshness problem of not knowing the version information of the latest content, 'CanBePrefix' helps to avoid an extra version requesting mechanism as follows: If interest $i_{3.1}$

```
i3.1: /newspaper/frontpage
      [MustBeFresh=true,CanBePrefix=true]
```

hits a content store with three data packets $d_{3.1}$-$d_{3.3}$,

```
Content Store:
d3.1: /newspaper/frontpage/v354    [FreshnessPeriod=0]
d3.2: /newspaper/frontpage/v360    [FreshnessPeriod=0]
d3.3: /newspaper/frontpage/v362    [FreshnessPeriod=500]
```

only $d_{3.3}$ will match the interest, without having the user to specify the version information. This seems to be an elegant way to request evolving content that is organized as versioned and immutable content. Unfortunately, this approach makes NDN non-deterministic. If the same interest is sent out again a few seconds later, version 362 is no longer 'fresh' and a newer and fresh version of the content may match the interest ($d_{3.4}$, continued after Figure 3.2).

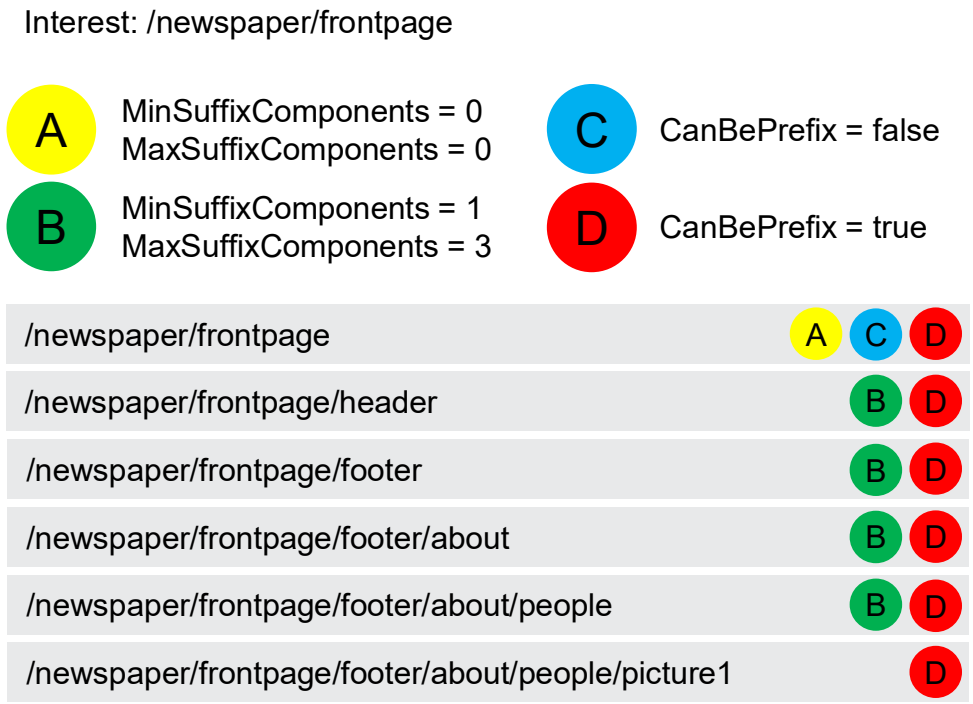


Figure 3.2 – Content Matching

'MinSuffixComponents' and 'MaxSuffixComponents' selectors were replaced by the binary 'CanBePrefix' switch during the change from NDN protocol specification version 0.2 to version 0.3. This figure shows which content names are match by interest `/newspaper/frontpage` with four different selector settings A, B, C, and D.

```
Content Store:
…
d3.3: /newspaper/frontpage/v362     [FreshnessPeriod=0]
d3.4: /newspaper/frontpage/v367     [FreshnessPeriod=320]
d3.5: /newspaper/frontpage/v370     [FreshnessPeriod=800]
```

Furthermore, when an even newer content $d_{3.5}$ is available at the producer, it is impossible for a requester to get hold of this newest content $d_{3.5}$ if 1) the requester does not know the exact version information and 2) a content store with still fresh content is in between requester and producer. The requester gets fresh but not latest content.

Name Space Design

Worse, if a name space is not carefully designed, a requester can be served with unexpected content. On purpose, Figure 3.2 shows a carelessly designed name space. Although no longer possible in NDN, an interest for `/newspaper/frontpage` with one additional name component can not only match (the intended) `/newspaper/frontpage/v370`, but also `/newspaper/frontpage/header` or `/newspaper/frontpage/footer`. Better designed names would be $d_{3.6}$ and $d_{3.7}$:

```
d3.6: /newspaper/2018-08-02/frontpage/v370/header/v120
d3.7: /newspaper/2018-08-02/frontpage/v370/footer/v82
```

However, this design is also not positive only. Imagine that 'frontpage' needed a change and was re-published as 'v371' ($d_{3.8}$).

```
d3.8: /newspaper/2018-08-02/frontpage/v371
```

The question is now what should be done with the unchanged 'header' ($d_{3.6}$) and 'footer' ($d_{3.7}$), or more generally with the whole subtree of 'frontpage/v370' ($d_{3.5}$). Although possible to further use it, consistent naming calls for re-publishing the same content under updated names $d_{3.9}$ and $d_{3.10}$:

```
d3.9: /newspaper/2018-08-02/frontpage/v371/header/v120
d3.10: /newspaper/2018-08-02/frontpage/v371/footer/v82
```

Moreover, even this slightly improved name space design is not suitable to work with the 'CanBePrefix' attribute. It is possible that $i_{3.2}$ gets satisfied with $d_{3.9}$ or $d_{3.10}$ and even $d_{3.6}$ or $d_{3.7}$ if they are still fresh.

```
i3.2: /newspaper/2018-08-02/frontpage/
      [MustBeFresh=true,CanBePrefix=true]
```

We do not argue that it is not possible to design better name spaces than that. However, it is an additional and challenging task. Furthermore, note that data packets flowing downstream must match PIT entries according to selector logic. Imagine data packet $d_{3.4}$ arriving at a node with given PIT entries $i_{3.3}$ - $i_{3.7}$:

```
Pending Interest Table:
i3.3: /newspaper/frontpage [CanBePrefix=true]
i3.4: /newspaper/frontpage/v367 [CanBePrefix=true]
i3.5: /newspaper/frontpage/v367 [CanBePrefix=false]
i3.6: /newspaper/frontpage/v367/favicon [CanBePrefix=true]
i3.7: /newspaper/frontpage/scripts [CanBePrefix=true]
```

PIT entries $i_{3.3}$ - $i_{3.5}$ must be satisfied while PIT entries $i_{3.6}$ - $i_{3.7}$ must not be satisfied. As mentioned above, NDN replaced 'MinSuffixComponents' and 'MaxSuffixComponents' through 'CanBePrefix' since version 0.3 (76). In contrast, CCN, originally also supporting 'MinSuffixComponents' and 'MaxSuffixComponents' in version 0.x, abandoned both attributes without substitution. Default and exclusive strategy is exact content matching (23).

Above all, not even exact content name matching is enough to be deterministic. This has to do with the way CCN and NDN try to enforce condition [B], so to frustrate the reuse of an already used name.

3.3.3 Implicit Digest Component and Content Object Hash

One way to make sure that an equal name cannot be attributed to a different content is to create a *unique* relation between name and content. Before looking at how CCN/NDN establishes uniqueness for data packets, uniqueness itself needs to be explicated.

Ambiguity & Uniqueness

Reasoning about the meaning of 'referential' is a good starting point to understand (un)ambiguity and uniqueness. The definition of a *reference* is the relation between two objects. The first object is the *name* for the second object, which itself is the so-called *referent* of the first object. Unambiguity is given if this relation is *unilateral definite from name to referent*. Hence, a name has at most

one referent, i.e. always the same. Nevertheless, a referent can have more than one name. Such relations are said to be *unambiguous*. In contrast, ambiguity means that the relation is *unilateral indefinite from name to referent*. Thus, a name may point to more than one referent. Such a relation is called *ambiguous*.

Uniqueness demands even more. It does not only require that one name points to no more than one referent, it does also require that no more than one name points to one referent. Hence, there is exactly one name for a specific referent, i.e. a *bilateral definite* relation. In other words, a unique name is a property possessed by one referent, but not by any other. Accordingly, a unique relation is always unambiguous while the opposite is not necessarily true. Figure 3.3 illustrates the different possibilities of relations.

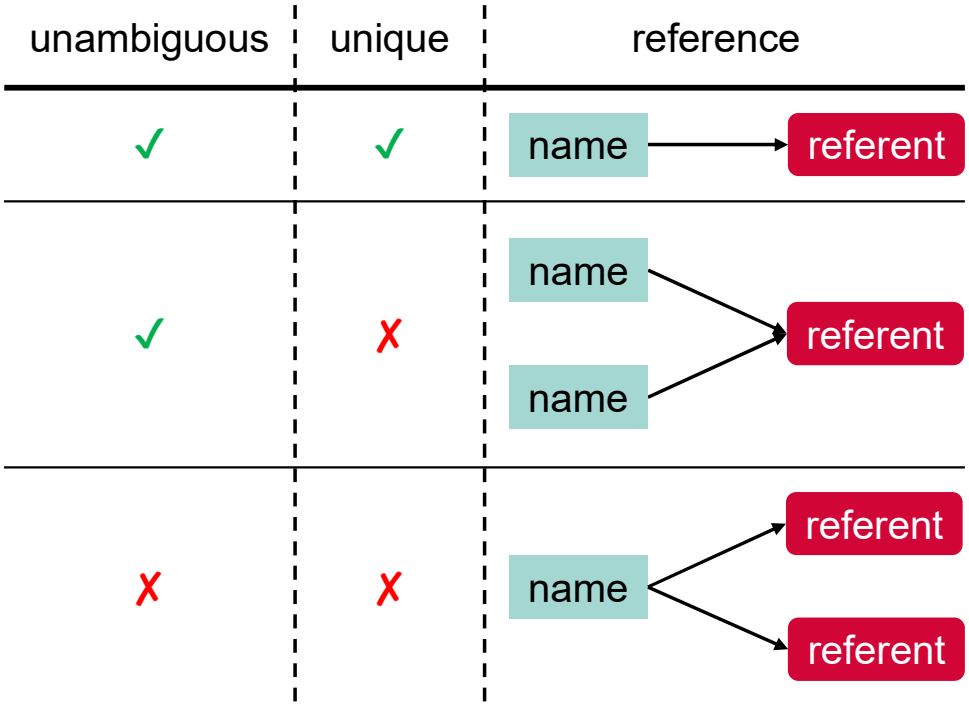


Figure 3.3 – Ambiguity & Uniqueness

Ambiguity rates if a name has one or more than one referent while uniqueness rates if a referent has one or more than one name.

NDN lists uniquely named (and immutable) data packets very prominently as one of their six deign principles (4). Uniqueness for every data packet is established by making names dependent on the content. Concretely, every name in an NDN data packet has a mandatory but 'implicit digest component'. It is the last component of the content name and obtained by computing the SHA-256 digest of all data packet bits (73). 'Implicit' means that the component is not explicitly part of the data packet and not transmitted along with the packet. The digest must be computed from everyone that wants to access it. CCN has a separate 'content object hash' (23), obtained likewise. In both cases, different content will generate different hashes. Hence, it makes data packets discriminable even if they have the exact same content name (neglecting the implicit digest component).

And here is exactly the problem: interests do not mandatorily contain the implicit digest component (or content object hash). The reason why is also comprehensible. Requesters would either need to guess the digest or, like the

freshness/versioning problem, they would have to retrieve the digest before requesting the content. NDN sees the use of digests in either requesting or excluding *specific* data packets (73). This clarifies that a name without the digest component is not enough to request a specific data packet. CCN speaks from the content object hash as 'restriction', instead of 'selector'. The renaming makes the use even clearer, i.e. restricting the number of matches. The conclusion remains: The name without implicit digest is not specific enough, although they claim that their exact matching is deterministic (23). Figure 3.4 shows two uniquely named NDN data packets. However, interest `/a/b/c` would match both packets because the implicit digest component is ignored if omitted in the interest.

NDN data packet 1	NDN data packet 2
Name (implicit digest in red): /a/b/c/4b1d83a79d3a1daadcaf00c884b47f5b8504f54aed65ddd6da02faeb298f4a47[27] **Content:** {"chapter":{"number":3,"title":"The Soft Spots"}}	**Name (implicit digest in red):** /a/b/c/941ac7116ef7fd866651ea3eb2e809f5adf73fa6efae15cb177b41bc7bafc35f[28] **Content:** <chapter><number>3</number><title>The Soft Spots</title></chapter>

Figure 3.4 – Uniquely Named Data Packets in NDN

Content names in data packets are unique by means of the implicit digest component that depends on the overall packet, including the content. Nevertheless, content matching considers this component only if explicitly given in the interest. Consequently, name resolution is not always deterministic in NDN. The same can be demonstrated for CCN.

However, note that uniqueness is overdoing for static content. Making sure that a content has only one name is not necessary. It is enough to ensure that no name resolves to two different contents (unambiguity). This seems easy for referentially transparent routines because an expression always resolves to the same result. At the same time, it gets evident that referentially opaque routines have problems with this requirement because an expression resolves to multiple results. Accordingly, the intermixing of side effecting, inherently non-deterministic routines and non-deterministic resolution mechanisms with the reuse of cached results and concurrent evaluations may lead to undesirable results. Therefore, we argue that it is necessary to differentiate between referentially transparent and referentially opaque requests and replies. Instead of classifying them only into RT and RO, we propose to use a finer-grained differentiation. The theoretical basis for this differentiation is developed in the following chapter.

[27] SHA-256 sum over a text file with content
"/a/b/c/{"chapter":{"number":3,"title":"The Soft Spots"}}"
[28] SHA-256 sum over a text file with content
"/a/b/c/<chapter><number>3</number><title>The Soft Spots</title></chapter>"

4. Equivalence Classes

Chapter 3 delivered good reasons why the handling of opaque expressions should be observed. This does not mean to exclude or neglect transparent expressions, but to have a clear notion of alternative result categories. What matters is the interpretation and determination of equivalences between names of requests (~name) and names of responses (~referent). This section outlines a non-exhaustive selection of different equivalences, from the referentially transparent 'syntactic equivalence' down to the referentially opaque 'adaptive equivalence'.

Within the space of referentially transparent expressions, defining more than only the syntactic equivalence has advantages like more flexible request to response matching, enabling more effective avoiding of re-computations. Within the space of referentially opaque expressions, avoiding all forms of parallelism and permanent caching would eliminate all concerns about correctness. However, this is conflicting with the envisaged distribution of computations by NFN. To maintain a certain level of parallelism and caching, some combinations of 'good-natured' referentially opaque expressions and adaptive determination of equivalences are identified during this chapter.

Informally, equivalence classes are attributes for expressions, indicating how they should be interpreted and matched against each other. This attribution is not evident from expressions themselves. A distinct expression can have several interpretations. The equivalences are summarized into an Equivalence Class System (ECS) at the end of this chapter.

Chapter	Section
Background	4.1 Syntactic Equivalence
The Soft Spots	4.2 Semantic Equivalence
Equivalence Classes	4.3 Adaptive Equivalence
Protocol & Architecture Adaptions	4.4 Equivalence Class System (ECS)
Evaluation	
Related Work	

TL;DR – Key Messages

- ✓ When requiring the syntactic equivalence, the name of a request must symbolically match the unique name of a response. Hence, a specific name resolves always to the same referent and inversely, a referent can be retrieved by exactly one name. This equivalence class implies deterministic results and no side effects. Requests can be aggregated and parallelized as well as results can be cached permanently and retrieved from caches respectively.
- ✓ The semantic equivalence enables to retrieve a referent by more than one name. It is enough if they are equivalent according to unambiguous rules. Results are still deterministic because identical names are still satisfied with identical referents. Calculi enable to perform computations on names and unambiguous aliasing tables enable content de-duplication, fallback names (for content) and function aliasing (for functions). Requests can be aggregated and parallelized as well as results can be cached permanently and retrieved from caches respectively.
- ✓ Adaptive equivalences are determined context-dependent and based on expression subclasses. Expression subclasses emerge from deconstructing results into return value and side effect.
- ✓ Ephemeral expressions have an ephemeral return value but no side effects. They cannot have no or a permanent return value. One name can point to several referents, depending on changed outer conditions, e.g. on time or an advancing file pointer. Requests can be parallelized and aggregated if explicitly desired. Return values cannot be cached permanently and not retrieved from caches respectively.
- ✓ Commutative expressions have an unproblematic side effect. The operation that causes the side effect must commute. Commutative expressions can have no, an ephemeral, or a permanent return value. Requests can be parallelized but not aggregated. Potential return values cannot be cached permanently and not retrieved from caches respectively.
- ✓ Sequence-idempotent expressions have an unproblematic side effect. The operation that causes the side effect must be commutative and idempotent. Sequence-idempotent expressions can have no or a permanent return value, but not an ephemeral return value. Requests can be parallelized and aggregated. Potential return values must be deterministic. They can be cached permanently and retrieved from caches respectively.
- ✓ Problematic expressions have a problematic side effect. The operations are whether commutative nor idempotent, e.g. writing an arbitrary value to a variable. Problematic expressions can have no, an ephemeral, or a permanent return value. Requests cannot be aggregated and must be evaluated sequentially. Potential return values cannot be cached permanently and not retrieved from caches respectively.

4.1 Syntactic Equivalence

Determining the equivalence of expressions is all about representation and interpretation of "symbols", combined with rules that define how they can be manipulated. For the most part, NDN names consist of generic name components. They are always interpreted syntactically. In other words, generic name components are seen only as identifiers, and never as a "number", "time" or anything else. Additionally, there is a limited set of other components with a distinct interpretation, e.g. the implicit digest component (a SHA-256 digest), the version name component (a number) or the timestamp name component (a Unix timestamp in microseconds) (77), (78). There is only one representation for a certain name. Rewritings are precluded, the sole equivalence that can be determined is the syntactic.

Syntactic equivalence is the strictest equivalence class. The relation is bilateral definite, i.e. unique. Therefore, it is referentially transparent. The name-to-referent ratio is 1:1. That is, on the creation of a content, exactly one unique name/identifier is assigned to it, and there is no other name to retrieve the content. CCN, as of version 1.0, insist on exact name matching. However, CCN packets contain another discriminative element, i.e. the content object hash. To be syntactically equivalent, the content object hash would have to match exactly, too. NDN does not insist on exact content name matching. To request it, 'CanBePrefix' must be (set to) false. However, to reach full syntactic equivalence, the implicit digest component must be set in every interest. This also applies to all other potential name components such as the version name component or the timestamp name component.

Mandatory digests are less tricky to handle if content was produced before requested. A requester may already know the digests, or alternatively, they can be retrieved by bundle. For example, a HTML file can contain names and digests of all integrated resources. Furthermore, if requested data is large and therefore chunked, it is possible to include the digests of the next few chunks in the current response. Including more than one digest enables to request chunks in parallel. However, dynamically created content, e.g. chat messages, are challenging. Even if the chat protocol reveals which name, version, and chunk must be assigned and requested next, the consumer cannot know the digest a priori. If there is more than one digest to a name, additional information is needed so that the user can choose the right digest. For example, such an additional information can be a producer identification.

Prefix registration authorities or mapping systems that equip local names with globally unique pre- or postfixes cannot guarantee for unique names as such. They allow to disambiguate two referents with the same name but different producers. Hence, they basically only prevent the abuse of names that should be unique.

For clarification, Figure 4.1 shows some example referents. Referents $a_{4.1}/a_{4.2}$ and $b_{4.1}/b_{4.2}$ use assigned prefixes to establish discrimination. Even if two referents occasionally carry the same payload ($a_{4.1}/b_{4.1}$), they differ in their names, and therefore also as a whole. Referents $a_{4.3}/a_{4.4}$ and $b_{4.3}/b_{4.4}$ demonstrate again the approach with implicit digests where referent discrimination is intact, too.

Figure 4.1 – Referent Discrimination

The payload alone is often too unspecific to uniquely discriminate referents. Putting names inside referents reduces but not eliminates ambiguous situations. However, an integrated hash over the whole data packet uniquely binds a name with its referent (referents $a_{4.3}$/$a_{4.4}$ and $b_{4.3}$/$b_{4.4}$). In contrast, unique prefixes (referents $a_{4.1}$/$a_{4.2}$ and $b_{4.1}$/$b_{4.2}$), assigned by an authority or mapping system, do not preclude ambiguous names.

Occasionally, it may appear that two producers produce a referent with the very same payload. If they use the same name by chance, referents will be identical, e.g. referents $a_{4.3}$ and $b_{4.3}$ in Figure 4.1. Nevertheless, this is not a problem. No one would even notice the "accident". In CCN/NDN, referents $a_{4.3}$ and $b_{4.3}$ would have different signatures because they have different producers. Signatures depend on producer *and* content object hash / implicit digest (see 2.2.1/Security). However, signatures are yet another level of discrimination. It is not their prior intent to discriminate contents but producers/provenances.

Although possible to realize unique relations, there is no benefit from enforcing it. Quite the contrary, it unnecessarily restricts the flexibility to have different names for the same referent. In subsection 3.3.3, we already insinuated that referential transparency does not require uniqueness. The following section 4.2 will delve into that space of referentially transparent but non-unique name relations.

4.2 Semantic Equivalences

In a referentially transparent relation, a referent can have more than one name. The name-to-referent ratio is *relaxed*, from a strict 1:1 relation towards a N:1 relation. N:1 relations are still transparent because identical names are still satisfied with identical referents. A name in a request may be syntactically equivalent to a name in the referent. However, it is enough if they are equivalent according to some rules.

The class of semantic equivalences is broad. As it was the case for syntactic equivalence, semantic equivalence is nothing that directly inhere in names. The decision (and responsibility) that two names are semantically equivalent and therefore point to referents with identical contents lies with whoever defined the rules. Likewise, it is the decision of every requester to accept or refuse a rule.

4.2.1 Calculi-Based Equivalences

Generally, a *calculus* is a method defined through a system of rules with the help of which certain mathematical problems can be treated systematically and solved automatically. Therefore, every calculus seems predestined to figure as rule set for the determination of an equivalence.

As the name suggests, λ-calculus is a calculus. However, it is not the only one. For example, *arithmetic* is also a calculus. Both calculi can be used to determine equivalences between names. However, instead of restricting an equivalence class to the four basic arithmetic operations addition, subtraction, multiplication, and division of integers and variables, it should rather include algebraic and other well-known rules such as the handling of roots and logarithms. Therefore, the class is generally named *mathematical* equivalence. The equivalence class based on λ-calculus is named λ-equivalence.

4.2.1.1 λ-Equivalence

Independent of NFN and its use of λ-expressions, names can generally be interpreted as λ-expressions. This requires detecting variables, λ-abstractions and applications. Therefore, only four discriminable "symbols" must be

equipped with a certain meaning. These are: the λ-operator "λ" to indicate a λ-abstraction, the dot "." to separate variables within a λ-abstraction, and opening and closing parentheses "(" / ")" to disambiguate expressions. In fact, any four other symbols could be used, these are just the common ones. There are no further meanings like "number" or "time" that must be detected.

Determination of Equivalence

An equivalence can be determined by bringing two expressions into canonical form and checking them for identity. An expression is in canonical form if it cannot be reduced any further and if it is unique, i.e. there is no other equivalent and irreducible expression than this one. Unfortunately, the confluence theorem of Church and Rosser says that irreducible expressions are unique, but they do not necessarily exist. Hence, the λ-equivalence can only potentially be determined. The confluent reduction rules in λ-calculus are α-conversion, β-reduction, and η-conversion.

With those ingredients, it is possible to determine the λ-equivalence, e.g. between λ-expressions $\lambda_{4.1}$ and $\lambda_{4.2}$ (these are the same expressions that were used in subsection 2.3.2):

```
λ4.1: /Name/Of/OuterFunction(
      /Name/Of/InnerFunction(/Name/Of/Data))

λ4.2:  (λxy.(x (y /Name/Of/Data))) /Name/Of/OuterFunction
                  /Name/Of/InnerFunction
```

Hence, a request carrying $\lambda_{4.1}$ as name can match a referent with name $\lambda_{4.2}$.

NFN uses λ-calculus to express computations, to compose results, to resume aborted computations, and to explore alternative resolution paths. However, NFN does not directly use λ-equivalence to match names from requests against names from result packets. One may think that NFN uses λ-equivalence because the FOX instruction (find-or-execute) maps a λ-expression to the CCN/NDN name of the result[29]. This seems like two names that are equivalent because they reference the same result/referent. Nevertheless, this is not the case. None of λ-calculus' reduction rules can transform $\lambda_{4.2}$ to $\lambda_{4.3}$.

```
λ4.3: /Name/Of/Result
```

The evaluation of binary function calls is not a λ-reduction step. Hence, an expression before and after reduction cannot be considered λ-equivalent. Technically, at the end of every computation, a new content is created with the canonical hash of the *initial computing configuration as name* and a *lookup function call for the latest computing configuration as data*. The latest computing configuration contains the name of the result stack on top of which the result can be found (3). Thus, the indirection hash finally resolves to a CCN/NDN name that is matched syntactically to a data packet's name.

Hence, NFN's FOX instruction works with the equivalence of computing configurations. However, this does not include bringing contained λ-expressions into canonical form. For example, $\lambda_{4.5}$ and $\lambda_{4.6}$ are syntactically different and therefore hash to different values, although they coincidentally reduce to the same value.

[29] Respectively to the CCN/NDN name of the updated computing configuration that holds the CCN/NDN name of the updated result stack that hold the result on top of it.

```
λ4.5:  (λx.((λx.x)x))x
λ4.6:  ((λx.(λx.x))x)x
       → x
```

Termination

The existence of a normal form is related to the termination of a computation. If no irreducible normal form of an expression exists, the computation will not terminate. *Untyped* λ-expressions as used by NFN may have no normal form. For example, the reduction process illustrated in Figure 4.2 does not terminate. Hence, no normal form is ever reached. Intermediate results are λ-equivalent to each other, but the final result does not exist. In typed λ-calculus, the canonical form of an expression can always be reached.

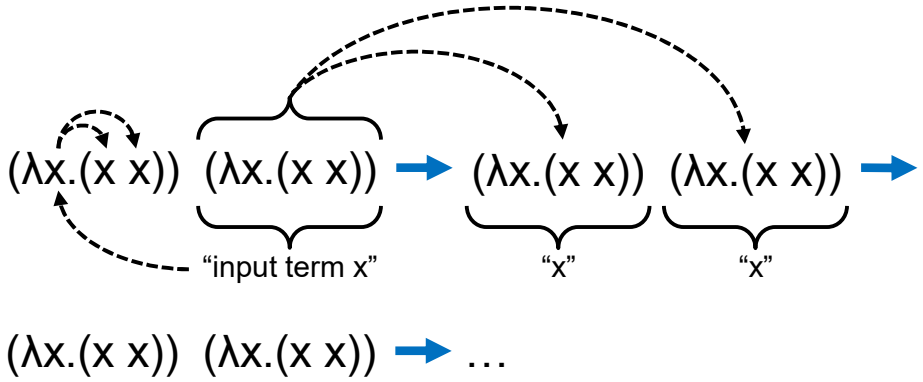


Figure 4.2 – Non-Terminating Reduction

The given λ-expression does not have an irreducible normal form. Therefore, its reduction/computation does not terminate. However, "reduced" expressions are λ-equivalent to each other. In this case, they are identical. Fat solid blue arrows indicate reduction steps while thin dashed black arrows help to follow the reduction process.

4.2.1.2 Mathematical Equivalence

λ-calculus is due to its repetitive character very inefficient in performing actual computations on the name level. Moreover, NFN never replaces parts of the name/λ-expression with results. Intermediate results are attributed a new and globally unique name that resolves to the intermediate result. Therefore, also efficiently computed results will never appear directly on the name level. However, having a global consensus on a notation for numbers and mathematical expressions enables *more efficient* computations *on* names instead of only doing computations *with* names. An example for such a global consensus can be that numbers are interpreted according to the decimal numeral system (base 10), the dot "." indicates the change from integers to decimals, the "+" symbol signify the addition etc. The advantage of predefined rules is that implementations can vary and/or progress, as long as rules are still adhered.

Mathematical rules are well known and therefore comprehensive for everyone. The relevant equivalence for this class is defined by the mathematical equality operator (=), which is not to be confused with the assignment operator (=) in programing languages like Java. The mathematical equality operator fulfills all requirements to be an equivalence relation in its pure mathematical sense. These requirements are:

- Reflexivity: A ≡ A
- Symmetry: A ≡ B if and only if B ≡ A
- Transitivity: If A ≡ B and B ≡ C then A ≡ C

Determination of Equivalence

When determining the mathematical equivalence, the emphasis lies on exact computations. Expression rewriting must base on rules that *symbolically* transform an expression into an equivalent one. For example, the system must symbolically recognize "0.1" as the 10th fraction of 1. The system must not internally represent "0.1" in binary floating-point representation for the reasons outlined in section 3.2. Systems that perform symbolic transformations are not obliged to cover the whole spectrum of known mathematical rules. They can be simplistic or advanced. For example, everyday programming languages can add or subtract integers with absolute precision and are therefore powerful enough by all means to ensure exact transformations. Hence, it is easy to determine the equivalence of '5+2' and '3+4' with absolute certainty by using integer data types, computing both expressions, and comparing the results, at least if numbers do not over- or underflow. An improved system may be able to interpret and process fractions, e.g. that 3/2 + 1/5 = 17/10. Specialized systems able to understand and manipulate mathematical expressions are called Computer Algebra Systems (CAS).

Canonical vs. Normal Form

There are two ways to check the equivalence of two expressions. Either both expressions are transformed to its *canonical form* and are then compared *syntactically*, or both expressions are transformed to its *normal form* and are then tested if their difference equals *zero*. The canonical form is to be understood as unique representation of an expression that cannot be further simplified, while the normal form is also understood to be irreducible, but not necessarily unique. Canonical forms need agreements on the expansion of polynomials, ordering of variables, handling of roots, and more. For a better understanding, consider the following example expressions:

[A] $(a+b)^2 = a^2 + a*b + b^2 = b^2 + b*a + a^2$
[B] $5*a*c = 5ac = c*a*5 = \cdots$
[C] $\sqrt{20} = 2*\sqrt{5}$
[D] $\sqrt{\frac{4}{7}} = \frac{\sqrt{4}}{\sqrt{7}} = \frac{2}{\sqrt{7}}$
[E] $\frac{1}{\sqrt{3}} = \frac{\sqrt{3}}{3}$
[F] $x^{-\frac{3}{2}} = \frac{1}{x^{\frac{3}{2}}} = \frac{1}{\sqrt{x^3}} = \frac{\sqrt{1}}{\sqrt{x^3}} = \sqrt{\frac{1}{x^3}} = x^{-1.5}$

All expression are valid canonical expressions, depending on the chosen agreement. For example, an agreement for [B] could be either "numbers before variables, explicit punctuation, alphabetic order" or "variables before number, explicit punctuation". A check on syntactic equivalence would yield 'false' or 'not equivalent'. However, regarding them as normal form expressions and checking their difference for zero would yield 'true' or 'equivalent'.

Canonical form, syntactic check: $5*a*c \neq c*a*5 \rightarrow false$
Normal form, zero check: $5*a*c - c*a*5 = 0 \rightarrow true$

Moreover, examples [C] to [F] show the dilemma of deciding when the canonical form is reached. The expressions can be interconverted in a cyclic manner. Nevertheless, Computer Algebra Systems hit the limit sooner as expected. In contrast to what their name suggests, Computer Algebra Systems not even guarantee the determination of equivalences for the whole class of algebraic expressions. As a repetition of section 3.2, algebraic expressions are finite and extend the set of arithmetic operations by raising to integer power and extraction of integer roots, i.e. by *rational exponents*. Despite that, Computer Algebra Systems guarantee the determination of equivalence only for *rational expressions*, i.e. expressions that can be expressed as a *rational fraction*. A rational fraction is a fraction of two polynomials, i.e. two expressions that only contain additions, subtractions, multiplications and non-negative *integer exponents*. Thus, roots and logarithms are not included. To give further reaching guarantees was proven impossible by Richardson (79). The theorem refers to algorithms in general, not to binary computations. However, knowing the transformation rules of roots and logarithms let CASs advance in regions beyond rational expressions. Concretely, they let CASs determine the equivalence of expressions as those of examples [C] to [F]. However, they cannot fully cover them. For example, a CAS in unable to determine the equivalence of

[G] $4^{\frac{3}{2}}$ and 8
[H] $\sqrt{4}$ and 2
[I] $\log_2 16$ and 4

because also CASs can only arithmetically approximate the expressions on the left side of [G], [H], and [I], to compare them with those on the right side. Lookup tables can cover some good-natured and frequent cases as those given above. However, lookup tables are only an aid but not a general solution. Nevertheless, the advantages of proven equivalence for the class of rational expressions and virtually infinite precision[30] for integer and floating-point arithmetic are undeniable. Whenever expressions are computationally complex, determining the equivalence to an already computed expression can significantly reduce the overall computing effort.

Search Strategy

Obviously, it cannot be beneficial to randomly explore the space of mathematically equivalent expressions. 'Obviously' because every mathematical expression has an unbounded number of equivalents. For example, 5 = 100/20 = 200/40 = 4+1 = 6-1 = 7-2 = 8-3 =…. Hence, when requesting name $n_{4.1}$, it does not make sense to also probe for names $n_{4.2}$, $n_{4.3}$, $n_{4.4}$, and so on.

```
n4.1: /Some/Specific/Function(4,1)
n4.2: /Some/Specific/Function(8:2,1)
n4.3: /Some/Specific/Function(4,10-9)
n4.4: /Some/Specific/Function(2*2,100^0)
```

The other way around, i.e. when initially requesting $n_{4.2}$, $n_{4.3}$, $n_{4.4}$, it makes sense to bring argument expressions into canonical form such as in name $n_{4.1}$. That way, all requests can be satisfied by a single referent, probably available from cache after a first request. However, the task of reducing argument expressions

[30] The precision of a CAS is *literally* infinite. However, as computers have a finite amount of memory, the precision of integers and floating-point numbers is only *virtually* infinite.

to canonical form can also be left to others. At the latest, the provider of `/Some/Specific/Function` must reduce the arguments. The result of the computation can then be returned together with the equivalence information. An advantage of this equivalence class is that it can be verified everywhere and at any time. Whenever there are doubts, an independent determination of the mathematical equivalence can be ordered.

Avoidance of Unintended Processing

Attention must be paid on correct indication of those parts of an expression that can be interpreted mathematically. Otherwise, unintended processing can appear. For example, the following interpretation from $n_{4.5}$ to $n_{4.6}$ should be avoided (most likely):

```
n4.5: /whitepages/search?country=us&phone=1-541-754-3009
n4.6: /whitepages/search?country=us&phone=-4303
```

Yet to mention is that globally unique and accessible named functions shall not have a direct mathematical meaning to a CAS, even if they have a clear mathematical background. Well-known function names like "log" or "sqrt" that has been previously agreed on may appear in expressions and can be interpreted mathematically. However, a CAS shall not determine a mathematical equivalence between $n_{4.7}$ and $n_{4.8}$, even if both names will ultimately resolve to the very same result.

```
n4.7: /Some/Specific/Function(/MathServiceXY/Log(2,16))
n4.8: /Some/Specific/Function(log(2,16))
```

4.2.2 Other Semantic Equivalences

Semantic equivalence cannot only be determined in the space of λ-expressions or real numbers and variables. Whenever a properly defined principle exists, it can be used for the determination of a semantic equivalence. For example, a time declared in Coordinated Universal Time (UTC) has a well-defined correspondent in Central European Time (CET), Eastern Standard Time (EST), etc. Referents $c_{4.1}$ and $c_{4.2}$ from Figure 4.3 exhibit two syntactically dissimilar names with identical information content. The names are equivalent in terms of time zone principles. Further details are discussed directly in the figure's legend.

Like the phone number example in subsection 4.2.1.2, it needs to be specified to which parts of an expression the transformation can be applied. Reasonably, semantically equivalent argument expressions should not lead to problems because they should not change the outcome of the routine call. Referents $c_{4.1}$, $c_{4.2}$, and $c_{4.3}$ from Figure 4.3 serve as positive examples. Referent $c_{4.3}$ demonstrates that an expression can have diverging interpretations. '0800+0100' mathematically yields '0900'. However, interpreted as date and time specification (e.g. (80), (81)), '0800+0100' is equivalent to '0800CET' or '0700UTC', meaning that the local time is 08:00 o'clock and the local time zone is one hour ahead of UTC time. Hence, to get UTC time, one must rather compute '0800-0100'.

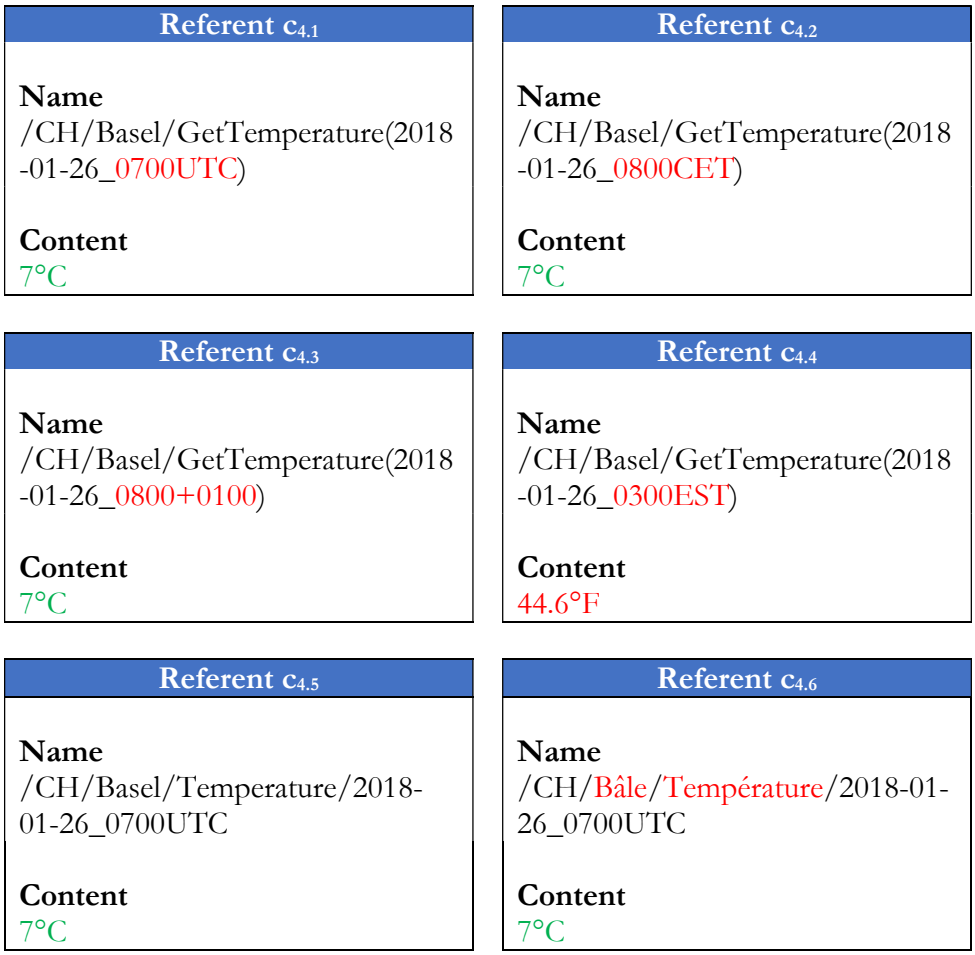


Figure 4.3 – Semantic Equivalence

All six data packets have semantically equal names. Equivalences are defined over different rule sets or principles. Referents $c_{4.1}$-$c_{4.4}$ make use of standardized notations for date and time, e.g. as defined in (80) and (81). Referent $c_{4.4}$ should also contain the content '7°C'. However, it was replaced on purpose with (the semantically equal) content '44.6°F' to discuss unsuitable function implementations. Compared to the 'function call'-like notation of previous names, the name in referent $c_{4.5}$ uses a rather static, CCN/NDN-like notation. Finally, referent $c_{4.6}$'s equivalence is defined over a linguistic dictionary.

Unit Transformations

Unit transformations are generally one of the most natural use cases for semantic equivalence. A positive example is given by the following two names $n_{4.9}$ and $n_{4.10}$ that are (should be) equivalent:

```
n4.9: /GetFahrenheit(7°C)
n4.10: /GetFahrenheit(280.15K)
```

Logically, both contained expressions yield 44.6°F. However, the indication of a unit is essential. Without a unit, it is unclear how to interpret the input *outside* of the routine declaration. The question is not if the function accepts both degree Celsius and Kelvin or just one of them as input. Calling the function with correct arguments is the duty of the requester. Once that the transformation rule between degree Celsius and Kelvin is known, as well as the

rule is known to be applicable to the argument of `/GetFahrenheit(x)`, two requests for $n_{4.9}$ and $n_{4.10}$ can be satisfied by the same referent. Obviously, the best case is if the implementation of `/GetFahrenheit(x)` can deal with either input. However, the transformation can also take place prior to the function call. A further example of semantic equivalence where units matter are trigonometric functions. Two notations are commonplace, i.e. angle and radian:

$\sin(90°) = 1$
$\sin(1/2\pi \text{ rad}) = 1$

Concerning ambiguity, this is a good example for the importance of units. For example, WolframAlpha[31] interprets arguments of trigonometric functions in a questionable way. Integers are interpreted as degrees but numbers with decimals are interpreted as radians, nota bene without a warning on an ambiguous input:

$\sin(5) \equiv \sin(5°) \approx 0.08716$
$\sin(5.0) \equiv \sin(5 \text{ rad}) \approx -0.95892$
$\sin(5.1) \equiv \sin(5.1 \text{ rad}) \approx -0.92582$

Generally, routines should be implemented in a way such that they do not change their behavior (and results) on the input of semantically equivalent inputs. Referent $c_{4.4}$ from Figure 4.3 serves as an example for what should not happen. In this case, the function `/GetTemperature(x)` is implemented such that it returns the temperature in the unit that is typical for the time zone associated with the input. Hence, it returns a result in degree Fahrenheit for EST times and a result in degree Celsius for CET times. Such a function implementation is unsuitable for semantically equivalent argument expressions.

Translations

However, semantic equivalence should not only be applicable on argument expressions but on all parts of an expression, i.e. also on identifiers. This enables more versatile naming patterns, e.g. like the name in referent $c_{4.5}$ from Figure 4.3. Its name should be identified as semantically equivalent to the names in referents $c_{4.1}$, $c_{4.2}$, and $c_{4.3}$. Data producers are not restricted to classic function call notation with an identifier (function name) and its arguments.

A special application of name component manipulations are linguistic "transformations", i.e. translations. In some cases, this might be very convenient. For example, names $n_{4.11}$-$n_{4.13}$ provide access to the same function in three different languages.

```
n4.11: /Math/Square(12)              [English]
n4.12: /Math/Quadrieren(12)          [German]
n4.13: /Math/ÉleverAuCarré(12)       [French]
```

Another example provides referent $c_{4.6}$ from Figure 4.3 that is a translation from referent $c_{4.5}$. The equivalence is determined by an English-to-French dictionary (Dictionary 1).

[31] http://www.wolframalpha.com/, tested on 30.01.2018: sin(5) resulted in 0.087156, i.e. sin(5°). sin(5.1) resulted in -0.925815, i.e. sin(5.1 radians).

```
Dictionary 1:
 GERMAN          FRENCH
 Basel       ↔   Bâle
 Temperature ↔   Température
```

A linguistic dictionary is nothing else that yet another rule set. However, in other cases, translations can lead to blurry situations. Names $n_{4.14}$-$n_{4.16}$ are perfect translations of each other.

```
n4.14: /Saint-Exupéry/LePetitPrince/Page1      [French]
n4.15: /Saint-Exupéry/DerKleinePrinz/Seite1    [German]
n4.16: /Saint-Exupéry/TheLittlePrince/Page1    [English]
```

Homonyms

Although it would be possible that these names point to the very same content, they rather yield different contents, namely the translated versions of the text. In this specific example of names $n_{4.14}$-$n_{4.16}$, syntactic equivalence is the way to go. Moreover, languages are inherently vague and therefore ambiguous. In linguistics, words with more than one meaning are called *homonyms*. For example, "spring" has 11 known meanings, "break" even 75 of them[32]. On the other hand, there are also specific words with just one known meaning, e.g. "blackboard" or "traditionally"[33]. Such words can easily lead to ambiguous situations because rewriting them can change the semantic meaning of an expression. Therefore, close attention must be paid to translations.

The challenge is to guarantee that no ambiguity is created whenever a new name is published subject to a dedicated dictionary. All possible translations must still point to the same referent. There is only one way to achieve this, namely, to restrict dictionaries such that every word can appear *at most* once. As a dictionary generally can be used in both directions, this restriction applies to the whole dictionary, not just to one side of it. For example, a UTC ↔ CET dictionary does not lead to problems because unambiguous. Dictionary 1 from above is also unambiguous. However, the German-to-English mini dictionary from below (Dictionary 2) has several problems. The German word "Brunnen" is both singular and plural of spring (~fountain). Moreover, it is the name of a village. On the other hand, "spring" can also mean the season "Frühling".

```
Dictionary 2:
 GERMAN       ENGLISH
 Brunnen   ↔  spring   [singular]
 Brunnen   ↔  springs  [plural]
 Brunnen   ↔  Brunnen  [place]
 Frühling  ↔  spring   [season]
```

Furthermore, a dictionary should be bound to a certain prefix. Otherwise, the dictionary may determine equivalences with names that are already in use elsewhere. The administrator of both prefix and dictionary is then responsible for avoiding ambiguous equivalences within its domain. Consequently, it is again not possible to just install rule sets in the wild and apply them at will. Interests must be marked in a way such that it is clear which rule set(s) may be applied. Finally, explicit attribution is again essential to verify translations.

[32] https://muse.dillfrog.com/lists/ambiguous (on May 31, 2018)
[33] https://muse.dillfrog.com/lists/wnt/specific/random (on May 31, 2018)

Aliasing

A more controllable way to work with translations is to define equivalences only between fully qualified names. This is, building up a list of *aliases*. Defining equivalences between unambiguous names will not cause non-deterministic results. The only requirement is again that names only occur in one *logical* table. 'Logical' means that an alias can technically appear in several tables. However, the union of those tables must refer to the same content. The issue is illustrated in Figure 4.4.

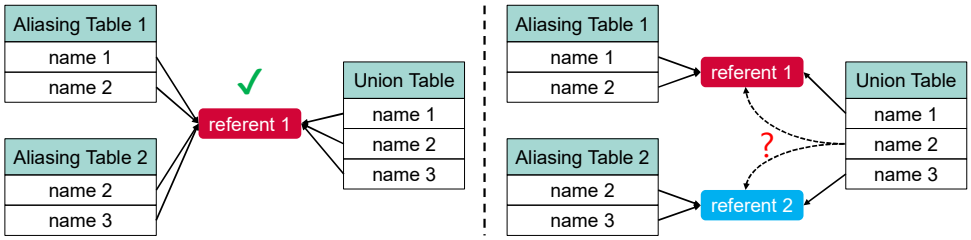


Figure 4.4 – Union of Aliasing Tables

The case on the left side of the dashed line is how it should be. 'name 2' appears in aliasing tables 1 and 2, once together with 'name 1' and once together with 'name 3'. This does not cause any problems if all four entries refer to the same referent. The union of aliasing tables 1 and 2 describes still a N:1 relation.

The two aliasing tables on the right side of the dashed line are valid if examined individually. All entries from one table refer to the same referent. However, because 'name 2' appears again in both tables, all entries of the union table must also refer to the same referent. As this is not the case, the right aliasing tables are incorrect and should be invalidated.

There does not have to be a mapping function or any further connection between aliases. For example, newspapers often buy articles or information from the same news agency (e.g. Schweizerische Depeschenagentur (SDA), Associated Press (AP), Bloomberg, Reuters) and publish them unchanged. This could be a global ranking of Universities, originally published by the SDA (referent $d_{4.3}$ in Figure 4.5), made accessible through two newspapers that offer the mentioned article under a specific name, e.g. in accordance with their column system (referents $d_{4.1}$ and $d_{4.2}$ in Figure 4.5). In terms of aliases, there are no such things as master and carbon copies. All instances are emancipated.

Content De-duplication

Normally, referents exist individually. However, aliasing can also be used for *content de-duplication* of (occasionally) identical contents. For example, referent $c_{4.2}$ in Figure 4.3 reveals a temperature of 7°C at the 26. January 2018 in Basel[34]. If the temperature is still the same 1 hour later or the next morning, it is enough to mark the three corresponding names $n_{4.17}$-$n_{4.19}$ as equivalent, and to store the content only once.

```
n4.17: /CH/Basel/GetTemperature(2018-01-26_0800CET)
n4.18: /CH/Basel/GetTemperature(2018-01-26_0900CET)  } 7°C
n4.19: /CH/Basel/GetTemperature(2018-01-27_0800CET)
```

(Continued after Figure 4.5)

[34] Randomly chosen temperature and date.

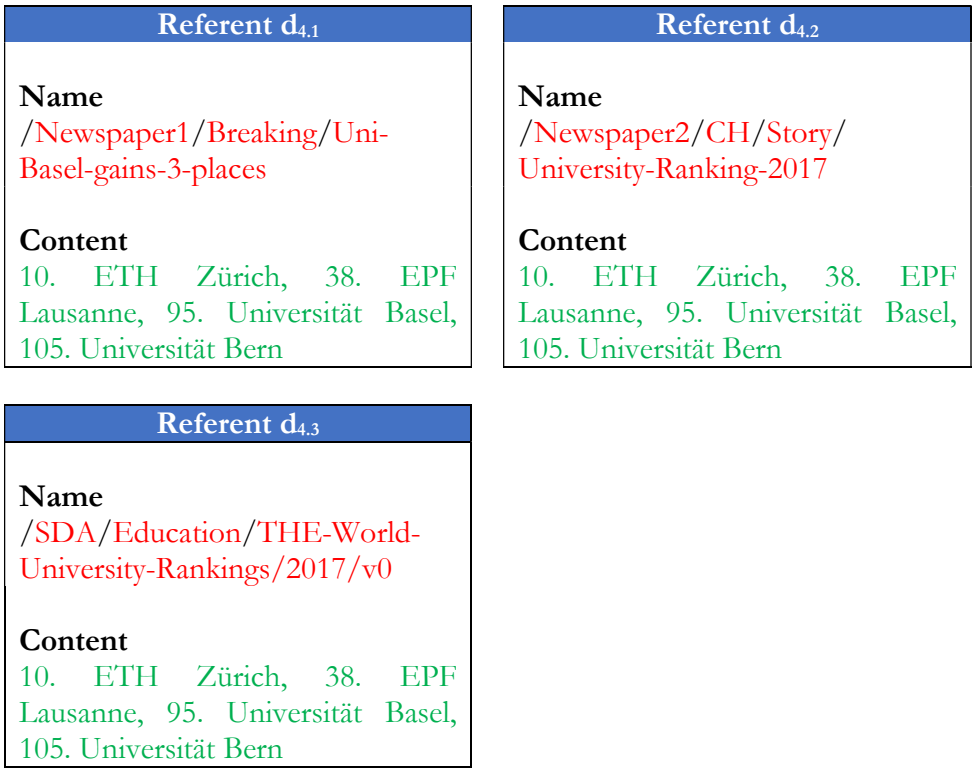


Figure 4.5 – Aliasing

Names of referents $d_{4.1}$ and $d_{4.2}$ are aliases of the name of referent $d_{4.3}$.

This also applies to function calls with a mathematical notion. For example, names $n_{4.20}$-$n_{4.23}$ occasionally yield the same result '4'.

```
n4.20: /MathServiceA/Fibonacci(3)
n4.21: /MathServiceA/Square(2)
n4.22: /MathServiceB/Log(2,16)
n4.23: /MathServiceC/Power(4,1)
```
} 4

Obviously, content de-duplication is more efficient the larger the content is compared to the name. Moreover, efficiency also depends on the implementation of the equivalence list. Instead of a simple table, a Bloom filter can be used. Adding an alias would then come at zero additional memory costs, unless the Bloom filter overflows and needs to be expanded. Nevertheless, as already mentioned on various occasions, primary goals of caching are the reduction of transmissions, lower latencies and an overall increase of network goodput, not memory savings.

Information Equivalence

However, referential transparency and the user's intention, no matter if explicit or implicit, require that the very same bits are delivered. It is not allowed that logically identical results are once expressed as ASCII char and once as 32 bit integer. For example, the number '4' above can also be written as Roman numeral 'IV', binary '100' or the English word 'four'. They have the same information content, but that is not enough. They are only equivalent if they are transcoded into the same representation.

Fallback Names

Furthermore, aliases can help in failover scenarios as fallback names. For example, if a request with name $n_{4.24}$ (for referent $d_{4.1}$ from Figure 4.5) cannot

be satisfied for whatever reason, the equivalence information may help to get hold of the content by alternatively searching for $n_{4.25}$ or $n_{4.26}$. This extends the failover capabilities of NFN that is "only" about finding new places to perform the computation, e.g. due to a broken link (38).

```
n4.24: /Newspaper1/Breaking/Uni-Basel-gains-3-places
n4.25: /Newspaper2/CH/Story/University-Ranking-2017
n4.26: /SDA/Education/THE-World-University-Rankings/2017/v0
```

Function Aliasing

The same applies to function calls. Two different implementations that return identical results for any possible input are semantically equivalent. Hence, the name of the first function is an alias for the name of the second function, and vice versa. In section 3.2, two examples showed how easy different implementations can lead to different results ('Repeated Rounding Problem' and 'Non-Associative Addition'). Despite this, there are also cases where reverting to alternative implementations is possible. For example, imagine different array sorting algorithms that lead to identical results in terms of ordering, accessible over names $n_{4.27}$-$n_{4.30}$.

```
n4.27: /CoolTools/FastArraySorting(arr)
n4.28: /CoolTools/MemoryEfficientArraySorting(arr)
n4.29: /CoolTools/RidiculouslyGoodArraySorting(arr)
n4.30: /CoolTools/ArraySorting(arr)
```

Given the same input array, four requests with names $n_{4.27}$-$n_{4.30}$ should return the same result, no matter if they are backed by the same or different implementations. If no result could be retrieved for an initial request, the system can fall back to an equivalent name that hopefully resolves. However, note the difference between determining equivalences and a λ-abstraction: If name $n_{4.30}$ would be a λ-abstraction, it would deterministically resolve to only one of the names $n_{4.27}$-$n_{4.29}$. Non-deterministic rewriting, e.g. in dependence of node-local conditions, cannot be reproduced in λ-calculus. Accordingly, if $n_{4.30}$ is a λ-abstraction, it cannot decide to locally call $n_{4.28}$ if free memory is limited, or $n_{4.27}$ if the CPU has a lot of idle time. A λ-abstraction sets the transition from one name to another name in stone. A node has no other choice than to follow the abstraction. In contrast, function aliases enable a node to rewrite a given name to a variety of names, constrained to the set of equivalent names. Consequently, every node can individually adapt its resolution strategy to current conditions, leaving the node a greater flexibility to optimize computations than just accepting or rejecting and forwarding a computation. However, a requester should be able to explicitly express its consent that a request is subject to function aliasing. Moreover, note that equivalence information cannot be extended or truncated without publishing a new rule set. The reason is that an independent node must be able to re-determine an equivalence for validation. Rule sets must not even contradict themselves, but if rule sets are not consistent, it may be impossible to re-determine a given equivalence due to lacking information.

So far, all classes were referentially transparent N:1 equivalences. As mentioned in the beginning of this chapter, equivalences should be determined adaptively on referentially opaque expressions. The next section explains why and delivers further helpful attributes.

4.3 Adaptive Equivalences

Contrasting referentially transparent expressions, referentially opaque expressions reflect N:M name-to-referent relations with M > 1. Whenever called, the result may differ from the result of a previous call. Hence, the easy recipe for treating referentially opaque expressions is to sequentially re-evaluate every call and not to cache its result. Accordingly, there seems to be no reason to determine an equivalence because their main intent is to aggregate requests and to match cached content as early and as often as possible. Even the aggregation of syntactically equivalent names seems to lead to problems because it may avoid individual requests being executed. For example, name $n_{4.31}$ should not be determined equivalent to name $n_{4.32}$ and it should not match the name in referent $r_{4.1}$, although they are all semantically equivalent.

```
n4.31: /TimeService/DateTimeNow("Basel")
n4.32: /TimeService/DateTimeNow("Basel")
r4.1:  /TimeService/DateTimeNow("Basel")
```

Adaptive Matching

A request for $n_{4.32}$ might have been issued later than or independently of another request for $n_{4.31}$. Therefore, the two requests should yield different results. For this example, a single 'referentially opaque' (RO) tag seems to be enough, preventing requests from being aggregated, results from being permanently cached or results from being delivered out of a cache respectively. However, it is obvious that requests not intended to aggregate must be distinguishable from each other. Additionally, their results must be request-specific, meaning that the determination of an equivalence must be *adapted* to components beyond the name, i.e. beyond function identifier and argument expressions.

Nevertheless, there are also special classes of RO expressions that can profit from cached results. In these cases, names are discriminative enough. Accordingly, a single RO tag is not significant enough to differentiate the cases. The determination of an equivalence must be *adapted* to current processing task and specific RO expression class. In other words, the determination of an equivalence is context-dependent. While this chapter assesses what classes are request-specific, the next chapter will focus on realization.

Result = Side Effect + Return Value

What matters for a finer-grained differentiation of RO classes, is a deeper understanding of what constitutes a result. So far, a result has been a *return value*. What is new in contrast to RT expressions, RO expressions can but not need to have a return value. Moreover, RO expressions can but not need to have a *side effect*. Side effects are also part of the result, i.e. the total of consequences of a routine call.

Permanence and Complexity

Additionally, the understanding of *permanence* in terms of a return value and *complexity* in terms of a side effect is essential for our purposes. Permanence in terms of return values is closely related to determinism of results. We use the term 'permanent' to linguistically demarcate it from 'deterministic'. Determinism concerns both parts of a result as it requires that there is no side effect. In contrast, permanence concerns only the return value and does not make any assumptions about side effects. It only says that a return value does not lose its validity, and therefore, can be cached permanently. Hence, RT expressions always have permanent return values. Non-permanent, i.e. 'ephemeral', return values are always associated with RO expressions.

However, RO expressions can also have permanent return values. Complexity in terms of side effects is the differentiation of expressions that can be evaluated concurrently and those who need to be evaluated sequentially. There is no need to wait for results of expressions with unproblematic side effects. The evaluation can be continued without having to worry about consistency. In contrast, waiting on results from expressions with problematic side effects is required. Otherwise, inconsistencies may appear.

Overall, this differentiation enables to identify 4 expression classes within the space of RO expressions that are of interest: ephemeral, commutative, sequence-idempotent, and problematic expressions. Table 4.1 shows a fine-grained classification of previously introduced equivalences, together with the new adaptive equivalences that will be introduced next.

Table 4.1 – Permanence and Complexity

This table categorizes different equivalence classes by means of permanence of their return value and complexity of their side effect. The rougher differentiation of referential transparency (white background) and referential opacity (grey background) is color-coded. The following short notations are used in the table: yes = allowed, no = not allowed or not applicable, aggregation = aggregation of requests allowed?, concurrent = concurrent processing allowed?, caching = permanent caching of results allowed?

<table>
<tr><th>return value / side effect</th><th>no</th><th>permanent</th><th>ephemeral</th></tr>
<tr><td>no</td><td>Effectless (RT) Expressions
• Aggregation: yes
• Concurrent: yes
• Caching: no</td><td>RT Expressions
• Aggregation: yes
• Concurrent: yes
• Caching: yes</td><td>Ephemeral Expressions
• Aggregation: opt-in
• Concurrent: yes
• Caching: no</td></tr>
<tr><td rowspan="2">unproblematic</td><td colspan="3">Commutative Expressions
• Aggregation: no
• Concurrent: yes
• Caching: no</td></tr>
<tr><td colspan="2">Sequence-Idempotent Expressions
• Aggregation: yes
• Concurrent: yes
• Caching: yes, if any</td><td></td></tr>
<tr><td>problematic</td><td colspan="3">Problematic Expressions
• Aggregation: no
• Concurrent: no
• Caching: no</td></tr>
</table>

1. **Ephemeral Expressions**
   - No side effect, ephemeral return value.
   - See subsection 4.3.1.
2. **Commutative Expressions**
   - Unproblematic non-idempotent side effect, permanence of return value insignificant.
   - See subsection 4.3.2.
3. **Sequence-Idempotent Expressions**
   - Unproblematic sequence-idempotent side effect, permanent return value.
   - See subsection 4.3.3.
4. **Problematic Expressions**
   - Problematic side effect, permanence of return value insignificant.
   - See subsection 4.3.4.

4.3.1 Ephemeral Expressions

Key properties of ephemeral expressions:
- Aggregation of requests: opt-in
- Concurrent processing: yes
- Permanent caching of results: no

The name giving property of this class comes from its results that are not stable over time, location, or any other outer condition that can change. Therefore, results of ephemeral expressions lose their validity sooner or later. This may happen frequently, e.g. when depending on time, or infrequently at an undefined point in time in the future, e.g. when depending on a hardware property that only changes with the replacement of the hardware. This property is responsible for that results must not be cached permanently. However, as ephemeral expressions do not have side effects per definition, there is, technically seen, no reason not to aggregate requests. Purity is an immediate guarantor for consistency because no state is manipulated. However, as it will be discussed below, there are good reasons why ephemeral expressions should not be aggregated *by default*.

Nevertheless, ephemeral expressions can always be evaluated in parallel due to their purity. For example, a request for a uniform random number on a given interval such as in $n_{4.33}$ is an ephemeral expression. A single requester that asks twice for a random number, or two requesters that each ask once, expect(s) two different results. Concurrency is responsible for that no guarantee is given on the perpetuation of any order of evaluation.

```
n4.33: /UniformRandomOnInterval(lowerBound,upperBound)
```

By all means, results are non-deterministic. To detail this property, imagine having two functions with names $n_{4.34}$ and $n_{4.35}$, and a third composed expression as in name $n_{4.36}$ that invokes the former two functions.

```
n4.34: /CurrentUnixTime()
n4.35: /Subtract(argExpr1,argExpr2)
n4.36: /Subtract(/CurrentUnixTime(),/CurrentUnixTime())
```

Marking $n_{4.34}$ in $n_{4.36}$ as ephemeral return value expression enables to evaluate argument expressions of $n_{4.36}$ concurrently and independent of each other. The whole expression $n_{4.36}$ is anyway referentially opaque. However, there is even no guarantee for the result to be negative or positive because this depends on which argument expression is evaluated first. Note that during the evaluation of $n_{4.36}$, the two requests for $n_{4.34}$ may be issued from one *or* two different requesters, e.g. from two different intermediate nodes. Likewise, it may appear that results are computed by one *or* two different providers. Moreover, all of them may be different from the provider of function $n_{4.35}$. In those cases, the provider of $n_{4.35}$ has no influence on any left-to-right evaluation preference of its argument expressions. Thinking of NFN, the provider may not even notice the arguments' opaque history as it only sees the newly created names that resolve to the results of the evaluated expressions.

Request Aggregation Revisited

In this place, one may argue that some requests for ephemeral expressions can be aggregated. For example, two simultaneous requests for $n_{4.34}$ can be aggregated and satisfied by a single result. The composed expression as in $n_{4.36}$ would then yield 0. However, strictly speaking, request aggregation is a very short-term cache because an aggregated request will be satisfied by a response to a prior request, although the result directly comes from the source and not from a cache. Request aggregation should therefore generally be forbidden for ephemeral expressions. Nevertheless, viewed from a different (more positive) angle, request aggregation is an *evaluation accelerator*. It allows a requester to retrieve responses in *sub-RTT*. On the one hand, one may rightly argue that such responses are "more accurate" or "rather real-time" than responses that took a full RTT. On the other hand, it is unclear how long the preceding request (that is responsible for the aggregation) already exists, which in turn can be countered again as irrelevant because waiting time is shorter. This, however, requires that no one satisfies such requests with old results on purpose. In order to counteract, results must contain a strictly monotonic increasing timestamp. With it, a requester can decide if the result is (sufficiently) new. If not, a requester must be able to circumvent aggregation in order to get a new result[35]. However, not only for this reason, request aggregation should be *opt-in*. An explicit label has the additional advantage that it eliminates all controversial discussion about expectations towards results of multiple routine invocations. Are two different responses expected, or are two identical copies fine, too? For example, request aggregation should clearly not be applied to expression $n_{4.33}$. Setting the label makes expectations clear. In other words, whenever a request is approved to be aggregated, it is an implicit renouncement on a request-specific result. Hence, the default setting should be 'no aggregation'. Requests that did not opt-in to aggregation, bypass matching of aggregated requests on their path towards the source. This implies that requests must be distinguishable from each other and results must be re-assigned request-specifically. Likewise, tagging can theoretically be used to jump over cache matching because, as mentioned above, results of ephemeral expressions are not cached permanently. However, this is not done for technical reasons (see 5.4.2).

[35] Note that this is the link to freshness and other matching criteria that we raised in subsections 3.3.1/3.3.2 and that we will continue in section 5.4.2.

4.3.2 Commutative Expressions

Key properties of commutative expressions:

- Aggregation of requests: no
- Concurrent processing: yes
- Permanent caching of results: no

Commutative expressions share, as their name suggests, the commutativity property. Hence, they can be evaluated concurrently and in arbitrary order. In contrast to ephemeral expressions where aggregation of requests *can* be possible, this is not the case for commutative expressions. It is required that every single call can unfold its side effect. Request aggregation would obviously frustrate this intention. Commutative expressions consist of nothing else than a single call to a function with the eponymous property. Prominent examples of commutative expressions are update operations on operation-based CmRDT replicas. In order to distribute updates and to make sure that every replica received the update, replicas must be aware of each other. This entails that replicas must be distinguishable. Therefore, they need unique names. Usually, the replica whose state is updated first, is then responsible to fan out the update to all other replicas. However, an example application will demonstrate that this responsibility can be outsourced to an arbitrary intermediary (section 6.2).

The trivial example is the increment of a replicated counter. To increment the counter, a requester must know at least the name of an increment function that is associated with the counter of one specific replica. It does not matter if this replica is local or remote to the user. The example works the same way in both cases. Imagine that 'Bob' knows the name of the increment function associated with replica number 1's counter. Accordingly, he can increment the counter by sending out a request with name $n_{4.37}$ and n = 1. Upon reception, replica number 1 applies the side effect locally, i.e. increments its local counter. As mentioned above, replica number 1 is then usually responsible to update all other replicas. In this example, these are replicas 2, 3, and 4. In order to do so, replica number 1 can send out requests for names $n_{4.38}$-$n_{4.40}$ with n = 1 *simultaneously*. As these requests carry different names, there is anyway no risk that they aggregate. However, if 'Alice' increments the same counter simultaneously with 'Bob' by issuing a request with $n_{4.41}$, requests $n_{4.41}$-$n_{4.44}$ will be simultaneously triggered with $n_{4.37}$-$n_{4.40}$. It is now important that $n_{4.37}$/$n_{4.41}$, $n_{4.38}$/$n_{4.42}$, $n_{4.39}$/$n_{4.43}$, and $n_{4.40}$/$n_{4.44}$ do not aggregate, despite syntactic equivalence. Like for ephemeral expressions, tagging them accordingly helps to jump over aggregation, but requests must again be made specifiable beyond the names of CmRDT increment expressions *only*.

```
n4.37: /Name/Of/CmRDT/Replica/1/Increment(n) [from 'Bob']
n4.38: /Name/Of/CmRDT/Replica/2/Increment(n)
n4.39: /Name/Of/CmRDT/Replica/3/Increment(n)
n4.40: /Name/Of/CmRDT/Replica/4/Increment(n)

n4.41: /Name/Of/CmRDT/Replica/1/Increment(n) [from 'Alice']
n4.42: /Name/Of/CmRDT/Replica/2/Increment(n)
n4.43: /Name/Of/CmRDT/Replica/3/Increment(n)
n4.44: /Name/Of/CmRDT/Replica/4/Increment(n)
```

In this example, it is also possible that individual replicas accumulate increments for a certain time span, e.g. for one minute, to then inform other replicas with a cumulative update, i.e. with n > 1.

Furthermore, it does not matter if commutative expressions have no return value, a permanent return value (e.g. "Successful increment"), or an ephemeral return value (e.g. "Successful increment with unique update code <abc123>"). Return values must not directly originate from a cache. Like aggregation, this would frustrate that every request unfolds its side effect. Accordingly, a return value, if there is any, is required not to be re-assigned to a request by the name of the CmRDT increment expression *only*.

4.3.3 Sequence-Idempotent Expressions

Key properties of sequence-idempotent expressions:

- Aggregation of requests: yes
- Concurrent processing: yes
- Permanent caching of results: yes, if any

Sequence-idempotence is a stronger property than idempotence only. Sequence-idempotence is given when an operation has an idempotent side effect that is stable over a sequence of other operations. Said differently, the relative position of each operation within a sequence does not influence the final outcome of the sequence. Two brief examples may help to grasp the difference between idempotence [1.] and sequence-idempotence [2.].

1. A simple light switch has two operations: 'switch on' and 'switch off'. Both operations are idempotent. For example, it does not matter if the light is switched on once or multiple times. The result is the same. The same is true for switching the light off. However, the operations are not sequence-idempotent because the final result depends solely on the last operation. This example makes clear that idempotent-*only* expressions cannot be aggregated. A slightly later operation may influence the outcome of the sequence. It also makes clear that idempotent-only expressions cannot be evaluated concurrently because the position in the sequence matters.

2. In contrast, a routine that remembers *distinct* museum visitors is sequence-idempotent because a visitor, once being remembered, cannot un-visit the museum. Additionally, it does not matter if the same visitor visits the museum just once or many times. He or she is remembered only once. Furthermore, it does not matter if a distinct visitor visits the museum early in the morning before everyone else, or late in the evening after everyone else. The outcome will be the same (he or she visited the museum). This example makes clear that *order and recurrence* of operations do not matter. Hence, they can be evaluated concurrently and they can be aggregated. Note, that if an initially switched off light only exposes a 'switch on' operation, this operation is sequence-idempotent, too.

To what it boils down is that sequence-idempotent operations must be *idempotent and commutative*. From CRDTs, we know a few such operations. Adding an element to a grow-only G-set, i.e. updating the G-set, is a sequence-idempotent operation. It does not even matter if the G-set is implemented as CmRDT or CvRDT. Example [2.] from above (museum visitations) could be

implemented as G-set. Once a visitor entered the museum, his or her unique visitor ID is added to the G-set. Invoking the function call from $n_{4.45}$ reflects this operation. In case of a CmRDT, the set addition is replicated through $n_{4.46}$-$n_{4.48}$. As in the replicated counter example, concurrent set additions do not lead to problems. Another attempt to add the same visitor over again ($n_{4.49}$) may be aggregated with the equivalent request $n_{4.45}$, or if already cleared, will have no further side effect.

```
n4.45: /Name/Of/CmRDT/Replica/1/Add(<visitorID>)
n4.46: /Name/Of/CmRDT/Replica/2/Add(<visitorID>)
n4.47: /Name/Of/CmRDT/Replica/3/Add(<visitorID>)
n4.48: /Name/Of/CmRDT/Replica/4/Add(<visitorID>)

n4.49: /Name/Of/CmRDT/Replica/1/Add(<visitorID>)
```

Merge operations of state-based CvRDTs are also sequence-idempotent because they are commutative and idempotent. Thus, once a replica updated its state ($n_{4.50}$), it can send its updated state concurrently to all other replicas for merger ($n_{4.51}$, $n_{4.52}$, $n_{4.53}$). Equivalent updates potentially occur rarely because the argument, i.e. the updated set, must be equivalent, too. However, if occurring, requests can be aggregated thanks to the sequence-idempotence property. Note that names of sequence-idempotent expressions alone are sufficiently specific to make aggregation decisions.

```
n4.50: /Name/Of/CvRDT/Replica/1/Add(<visitorID>)
n4.51: /Name/Of/CvRDT/Replica/2/Merge(<updatedSet>)
n4.52: /Name/Of/CvRDT/Replica/3/Merge(<updatedSet>)
n4.53: /Name/Of/CvRDT/Replica/4/Merge(<updatedSet>)
```

Note that in case of CvRDTs, the update function needs only to monotonically increase the internal state (see subsection 2.5.1.2). It must not necessarily be commutative. However, the addition to a G-set is commutative because it bases on the union function that is commutative (63). Therefore, $n_{4.50}$ is commutative and can be parallelized. If the update function is only monotonically increasing but not commutative, it must be marked as problematic expression (see subsection 4.3.4), and therefore must be evaluated sequentially. However, this only concerns the update function. Subsequent merge functions are always commutative, associative, and idempotent.

Concerning return values and their potential permanent caching, a few things are to note. Sequence-idempotence does not say that an expression is not re-evaluated when called a second time. It says, if called a second time, no further state change occur. Therefore, any further evaluation can be avoided. Hence, in the optimal case, a sequence-idempotent expression produces (besides the side effect) a permanent return value that subsequently gets cached. The permanently cached return value can then satisfy later calls to the same function that would unfold no effect at all. Like for aggregation, the name of a sequence-idempotent expression alone is sufficiently specific to match cached results and to re-assign results to requests. Satisfying such calls as early as possible saves network resources. Subsection 5.4.2 deepens technical aspects and possibilities of return values for sequence-idempotent expressions. Consistency is preserved in any case, even if no return value prevents the re-evaluation of new equivalent calls. However, sequence-idempotent expressions

must not have ephemeral return values. This would cancel out any additional advantages arising from sequence-idempotence over commutative expressions.

4.3.4 Problematic Expressions

Key properties of problematic expressions:

- Aggregation of requests: no
- Concurrent processing: no
- Permanent caching of results: no

Problematic side effects, i.e. side effects that do not commute and that are not idempotent, are the worst case in terms of distributed computations. There is no other choice than evaluating them according to some synchronization technique, e.g. locks (mutex), spinlocks, semaphores, readers-writers locks, or more sophisticated mechanisms like read-copy-update. Generally, these mechanisms organize the access to a *variable shared resource (~mutable state)*, often referred to as *critical section*. A write request for a problematic expression must first *acquire* the shared resource, i.e. the permission to write to the variable. As soon as the access to the shared resource has been granted, the requested expression can be evaluated. Thereafter, the shared resource should be *released* such that other requests can access the resource, too. This leads to sequential processing of requests as discussed and may introduce noticeable delays during the evaluation of expressions. The distribution of calculations thus becomes a disadvantage, a local execution an advantage. Consequences are clear and far-reaching: no aggregation and concurrent evaluation of requests, as well as no caching of results, independent of a potentially permanent return value.

Unfortunately, problematic expressions, mostly in conjunction with ephemeral return values, are probably the most common form of consumed content nowadays. For example, financial transactions such as equity trading ($n_{4.54}$) are inherently problematic. In order to trace back every transaction, they can be equipped with a unique transaction identifier. Side effects, i.e. removing shares in one depot, adding them to another depot, and writing related transaction information to a log, are problematic. Moreover, the three side effects form an atomic transaction. However, *atomicity* of several (side-effecting) expressions must be handled separately on top of equivalence classes, e.g. with a three-phase commit protocol (3PC). Additionally, the return value is ephemeral if a unique transaction identifier is included. However, this has no further consequences. Further examples are filter bubbles or any kind of personalized content that was created with reference to a recommender system ($n_{4.55}$, $n_{4.56}$). Recommender systems are often part of online stores and movie streaming services. The pervasive use of trackers throughout the web are responsible for that almost every request has side effects. Due to this personalization, both side effects and return values are highly variable.

```
n4.54: /TheFancyBank/TradeShares
       (sellersDepotNo, buyersDepotNo, shareName, amount)

n4.55: /TheSearchCompany/GetResults
       ("AdaLovelace",browserID)

n4.56: /TheSocialNetwork/LoadNewsfeed(userID, password)
```

Apart from not using problematic expressions, like all purely functional approaches are doing it, there is no way to mitigate the negative effects they have on the distribution of computations. It only remains to recognize, tag and handle them correctly such that no inconsistencies appear. Aggregation can be jumped over and the determination of equivalences must be request-specific when it comes to re-assignment of results to requests. Cache matching is again unnecessary in theory but should not be avoided for technical reasons (see 5.4.2).

4.4 Equivalence Class System (ECS)

The synthesis of gained insights is a new classification system for the attribution of expressions. It is tailor-made for NFN-like name-based distributed computations in information-centric networks. We call the classification system *Equivalence Class System* (ECS). Table 4.2 illustrates the detailed constellation of the Equivalences Class System.

Table 4.2 – The Equivalence Class System (ECS)

This table summarizes equivalence classes that have been introduced in this chapter. The four expression types of the adaptive equivalence are listed individually. The list is not final or exhaustive and can be extended at any time. Short and distinct ECS tags are chosen for convenient use. However, as the list may expand, they may change, too, in order to preserve distinctiveness.

	Syntactic	Semantic	Ephemeral	Commutative	Sequence-Idempotent	Problematic
Name-to-Referent Ratio	1:1	N:1	N:M	N:M	N:M	N:M
Referentially Transparent	✓	✓	✗	✗	✗	✗
Referentially Opaque	✗	✗	✓	✓	✓	✓
Unique	✓	✗	✗	✗	✗	✗
Unambiguous	✓	✓	✗	✗	✗	✗
Pure (Nullipotent)	✓	✓	✓	✗	✗	✗
Impure (Side Effect)	✗	✗	✗	✓	✓	✓
Unproblematic Side Effect	✗	✗	✗	✓	✓	✗
Problematic Side Effect	✗	✗	✗	✗	✗	✓
Commutative	✗	✗	✗	✓	✓	✗
Idempotent	✗	✗	✗	✗	✓	✗
No Return Value	✗	✗	✗	✗/✓	✗/✓	✗/✓
Permanent Return Value	✓	✓	✗	✗/✓	✗/✓	✗/✓
Ephemeral Return Value	✗	✗	✓	✗/✓	✗	✗/✓
ECS Tag	syn	sem	eph	com	seq	prb

The ECS summarizes different result qualities and reveals their particularities. We supplied each class with a unique *ECS tag*. These tags refer to the corresponding type of equivalence and are used to attribute expressions in names of requests and responses. The following chapter 5 will point out how to integrate these theoretical considerations into the existing generalized CCN/NDN execution model as described in subsection 2.2.2. This includes extensions of protocol and architecture.

5. Protocol & Architecture Adaptions

Performing computations in an information-centric network, especially if they are opaque, entails new challenges that must be solved. Challenges are either related to the protocol, e.g. how to proceed with interests that qualify for semantic equivalence, or related to the architecture, e.g. additionally needed packet fields or how to attribute expressions.

The section starts by defining a notation for attribution and by reasoning what it means to compose expressions of different classes (section 5.1). Forwarding is explained consistently for all equivalence classes because it works the same for all of them (section 5.2). Afterwards, referentially transparent (section 5.3) and referentially opaque (section 5.4) expressions are examined independently of each other towards required protocol and architecture adaptions. To close this chapter, the generalized CCN/NDN execution model, as presented in subsection 2.2.2, will be developed in direction of an extended execution model (section 5.5) that covers the handling of all newly introduced equivalence classes.

Chapter	Section
Background	5.1 Attribution and Composed Expressions
The Soft Spots	5.2 Forwarding & PIT Timings
Equivalence Classes	5.3 Working with Referentially Transparent Expressions
Protocol & Architecture Adaptions	5.4 Working with Referentially Opaque Expressions
Evaluation	5.5 Extended Execution Model
Related Work	

TL;DR – Key Messages

- ✓ Equivalence class attribution does not happen at the interest level, but on the (sub)expression level. Therefore, attributes must be integrated directly in names/expressions. The following ECS tags are used for the attribution of (sub)expressions:
  `[syn]`, `[sem,X]`, `[eph]`, `[com]`, `[seq]`, `[prb]`
- ✓ An additional aggregation attribute enables to aggregate ephemeral expressions.
- ✓ Equivalence classes supersede each other in composed expressions according to the following rule. left < right means that right supersedes left. left = right means that no superseding occurs.
  `[syn]=[sem]=[seq] < [eph,aggr] < [eph]=[com] < [prb]`
- ✓ Interests are always forwarded in direction of the rewritten expression.

- ✓ Result packets for expressions that have been aggregated due to equivalences must be re-packed with equivalent names and re-signed by the node that determined the equivalence.
- ✓ Requests for `[eph]`, `[com]`, and `[prb]` expressions must be satisfied with request-specific (~per-request) results. In order to disambiguate homonymous requests in the PIT and to re-assign results request-specifically, interests and data packets must include another discriminative element than the name. We propose to use the nonce value.
- ✓ To enable reliable broadcast, all types of results need to be cached temporarily, also those of expressions with ephemeral return values or side effects.
- ✓ Garbage collection can be improved by observing the following eviction order (from evict first to evict last):
  `[eph],[com],[prb]→[eph,aggr]→[seq]→[syn]→[sem,X]`
- ✓ Mutable states are bound to an entity and can only be accessed via pinned associated functions.

5.1 Attribution and Composed Expressions

Notation

A first challenge is how to attribute expressions such that their class affiliation can be recognized throughout the network. This is an essential precondition for interest aggregation, permanent caching and content matching.

There are two things to consider:

1. An expression contained in an interest can be composed of several subexpressions, all of them possibly associated with a different equivalence class. Hence, class attribution does not happen at the interest level, but on the (sub)expression level. It is therefore not enough to integrate a single 'equivalence class' field in interests. Attributions must be integrated in names/expressions.
2. Obviously, side effects and return values appear in reaction to routine calls. Therefore, it seems evident to simply attribute individual calls. However, enabling the attribution of anonymous expression blocks eases the use of names. Not needing to declare a specific named function for every operation is a major benefit of the semantic equivalence class. The information about applicable rule sets is enough to reduce anonymous parts of an expression.

To give some initial examples, a few new notation rules need to be introduced.

- For **routine calls**, the equivalence class is inserted between brackets at the end of the routine call, i.e. after the (maybe empty) list of arguments between parentheses.

  → `/Routine/Call(argumentList)[<eqClass>]`

- When requesting static content, the empty list of arguments should be omitted because it enables to syntactically distinguish them from routine calls. Requesting static content can be seen as calling a generic 'read' function that is associated with the content object. However, a request for static content implies that the result is immediately available (if it exits at all), while a function call *may* imply some prior time-consuming computations. Differing these two cases is relevant for timing issues that will be discussed in section 5.2.

  → `/Some/Static/Content[<eqClass>]`

- Equivalence classes are indicated by **ECS tags**. Thus, the relevant attributes are

  → `[syn]`, `[sem]`, `[eph]`, `[com]`, `[seq]`, `[prb]`

- Brackets indicate a **list of attributes**, the equivalence class is just one of them. Whenever needed, additional attributes can be appended to the list, separated by commas.

  → `[<eqClass>, <attr2>, <attr3>, …]`

  For example, to express opting-in for interest aggregation of ephemeral expressions, an optional interest aggregation tag 'aggr' can be appended.

  → `/CurrentUnixTime()[eph,aggr]`

  Another example that requires an additional attribute is the semantic equivalence class that needs a (globally) unambiguous rule set identifier.

  → `/Name/With/Aliases(argumentList)[sem,<ruleSetId>]`

  A brief note on rule sets: Rule sets can feature many appearances. They can be associative arrays, rewrite rules such as those from λ-calculus, or parsers, e.g. to grammatically detect mathematical expressions. Moreover, the keys of an associative array may not only be a string to search and replace, it may be as well a rule itself, e.g. a regular expression that defines a search pattern.

  Rule set identifiers behave the same as CCN/NDN names regarding unambiguity. It suffices if they are unambiguous within a specific network. If the application space is a private network (~intranet), locally unambiguous identifiers are acceptable. Only on a global scale, i.e. the public Internet, they need to be globally unambiguous.

- If no equivalence class is indicated, the **worst case** must be assumed, i.e. a problematic expression.

  → `[]` ⇔ `[prb]`

With the given fall back rule, the equivalence class of an outer call does not transitively apply to it arguments, implying that argument expressions should be attributed individually. Hence,

```
/Outer/Function(/Inner/Function())[syn]
```

is in fact

```
/Outer/Function(/Inner/Function()[prb])[syn]
```

However, note that this fallback rules apply only to routine calls. If an argument expression does not contain any routine calls, the syntactic equivalence class can be assumed. Consequently, argument expressions without routine calls do not taint the overall call, i.e. they do not supersede the outer attribution. More details on equivalence class superseding follows later in this section.

- Brackets '[…]' delimit the **application sphere** of an attribute within anonymous parts of expressions. For example, it is not allowed to mathematically interpret the arguments of expression $e_{5.1}$.

```
e5.1: /An/Opaque/Function(2+2,4*10)[prb]
e5.2: /An/Opaque/Function
          ([2+2][sem,1],[4*10][sem,1])[prb]
e5.3: /An/Opaque/Function([4][sem,1],[40][sem,1])[prb]
```

In contrast, expression $e_{5.2}$ enables the mathematical interpretation of its arguments, e.g. by a computer algebra system. Imagine that the rule set with identifier '1' represents a rule set that can detect and interpret mathematical expressions (see *"Difficult" expressions* in section 3.2). Hence, an equivalence of expression $e_{5.2}$ and $e_{5.3}$ may be determined.

However, expression parts outside of brackets are subject to the above fall back rule. For example, expression $e_{5.4}$ is equivalent to expression $e_{5.5}$ but not equivalent to expression $e_{5.6}$.

```
e5.4: /An/Opaque/Routine
          (3+[2+2][sem,1],[4*10][sem,1])[prb]
e5.4: /An/Opaque/Routine
          (3+[4][sem,1],[40][sem,1])[prb]
e5.6: /An/Opaque/Routine
          ([7][sem,1],[40][sem,1])[prb]
```

- A fully reduced semantic expression should be re-attributed to `[syn]`. This signifies to subsequent nodes that no further processing is possible. Hence, they only must check for syntactic equivalences.

→ `[2+2][sem,1]` ⇒ `[4][sem,1]` ⇒ `[4][syn]`

Composed Expressions

One way to compose expressions of different equivalence classes is *nesting*, for example as done in above expressions $e_{5.2}$-$e_{5.6}$. As broached in section 4.3, calling a referentially transparent expression with referentially opaque argument expressions results in a composed expression that altogether is opaque. However, as demonstrated, opacity alone does not immediately imply the abandonment of all advantageous property. It is again the finer-grained differentiation into equivalence classes that matters. The question is how equivalence classes of inner and outer expressions influence the overall equivalence class of the composed expression. This is important because requesters and relaying nodes need to know how to treat the composed expression. For example, imagine an addition that is referentially transparent, with one summand being referentially opaque, e.g. a call to get the current Unix timestamp. Expressions $e_{5.7}$ and $e_{5.8}$ are two flavors how to implement the given functionality. Expression $e_{5.7}$ uses a specific implementation of the addition, while expression $e_{5.8}$ uses a rule set that specifies the semantic meaning of the "+" operator.

```
e5.7: /Add(/CurrentUnixTime()[eph],1000)[syn]
e5.8: [/CurrentUnixTime()[eph] + 1000][sem,1]
```

Irrespective of the implementation, both expressions should not be matched against the content store, although the outer function is referentially transparent and therefore would allow the retrieval of cached results.

Equivalence Class Superseding

The decision which equivalence class supersedes another equivalence class is relatively simple. As a rule of thumb, it holds that the "bad guy" prevails over the "good guy". Taking the information from Table 4.1 and Table 4.2, equivalence classes can be sorted from "good" to "bad", resulting in the reconditioned overview of Figure 5.1.

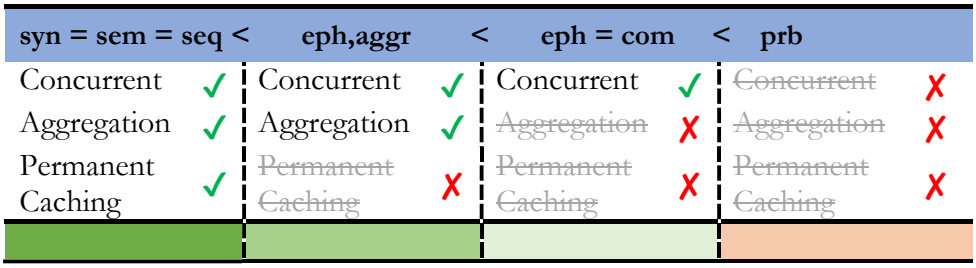


Figure 5.1 – From "Good" to "Bad" Equivalence Classes

left < right means that right supersedes left. left = right means that no superseding occurs. Moreover, this figure illustrates the loss of desirable properties from "good" to "bad" equivalence classes. The color codes at the bottom of this figure are used in Table 5.1 to refer to the corresponding properties.

With the explanations for superseding of Figure 5.1, Table 5.1 can be derived. It shows the resulting equivalence class that applies to a composed expression for all distinct pairs of equivalence classes. In addition to Figure 5.1, Table 5.1 differentiates between outer and inner, i.e. nested, equivalence class. This differentiation reveals what decisions are to be made in cases with different but not superseding equivalence classes.

Table 5.1 – Mutual Equivalence Class Superseding for Composed Expressions

Composed expressions that are sent out by requesters or relayed by intermediate nodes must be treated according to this table. It defines mutual superseding for all distinct pairs of equivalence classes. The cell coloring is according to the color codes introduced in Figure 5.1.

The following pairs do not supersede each other: (*1) syn / sem, (*2) syn / seq, (*3) sem / seq, (*4) eph / com. Hence, the outer attribute remains for the composed expression. The inner attribute is not lost and still applies to the inner part. However, the requester of the composed expression must treat the composed expression according to the outer attribute.

Moreover, the compositions marked with (*5) can be treated as follows: syn / seq → syn, sem / seq → sem, although these composed expressions have side effects. However, a result that is available in a cache implies that the side effect already took place and that the result is now permanently valid.

outer \ inner	syn	sem	seq	eph,aggr	eph	com	prb
syn	syn	syn[*1]	syn[*2*5]	eph,aggr	eph	com	prb
sem	sem[*1]	sem	sem[*3*5]	eph,aggr	eph	com	prb
seq	seq[*2]	seq[*3]	seq	eph,aggr	eph	com	prb
eph,aggr	eph,aggr	eph,aggr	eph,aggr	eph,aggr	eph	com	prb
eph	eph	eph	eph	eph	eph	eph[*4]	prb
com	com	com	com	com	com[*4]	com	prb
prb	prb	prb	prb	prb	prb	prb	prb

Implicit Analysis or Wrapping

The relevant equivalence class can be derived implicitly through analyzing the expression with the superseding table from above, or explicitly though wrapping. For example, $e_{5.9}$ is the wrapped version of $e_{5.7}$ and $e_{5.10}$ is the wrapped version of $e_{5.8}$.

```
e5.9: [/Add(/CurrentUnixTime()[eph],1000)[syn]][eph]
e5.10: [[/CurrentUnixTime()[eph] + 1000][sem,1]][eph]
```

Wrapping can be omitted if the outermost equivalence class is relevant, i.e. not superseded from the equivalence class of an argument expression. Implicit analysis is computationally more expensive for every involved node while explicit attribution increases the number of transmitted bytes.

Note that the concept of function aliasing can be applied to RO expressions. As it applies to RT functions (see subsection 4.2.2), replacing a call to a RO expression through a call to another RO expression is valid, given that they implement the same logic. It applies because function aliasing only influences the name side of the name-to-referent relation. A name is exchanged through another name, not constraining the referent side to be immutable. Therefore, adaptive equivalence classes have a N:M name-to-referent ration and not only a 1:M ratio. However, the concepts of fallback names and content de-duplication require results that do not lose their validity. Their retrieval is a transparent operation and therefore cannot be transferred on RO expressions. It is possible to implement function aliasing for RO expressions either by introducing rule set identifiers for all adaptive equivalence classes or by

wrapping. Imagine that a rule set with identifier '2' defines three translations as being equivalent:

```
Rule set '2' (aliasing table):
/TheSearchCompany/News
/TheSearchCompany/Neuigkeiten
/TheSearchCompany/Nouvelles
```

To make use of rule set '2', a requester can either ask for $e_{5.11}$ or $e_{5.12}$. The overall expression remains opaque. Again, this can be derived trough implicit analysis or through another explicit wrapping, as shown in $e_{5.12}$.

```
e5.11: /TheSearchCompany/News(browserID)[prb,2]
e5.12: [[/TheSearchCompany/News(browserID)[prb]][sem,2]
       ][prb]
```

5.2 Forwarding & PIT Timings

A question that can be answered generally is how different expression classes are matched against the forwarding information base (FIB), i.e. how equivalence classes influence forwarding decisions. The short answer is: not at all.

The long answer: FIB matching is not adapted to any equivalence class. Interests are always forwarded in direction of the rewritten expression. This is explicitly also the case for aliasing. For example, take the aliasing table from below with rule set identifier '3' and expressions $e_{5.13}$ and $e_{5.14}$.

```
Rule set '3' (aliasing table):
/The/Original/Name
/The/Rewritten/Name

e5.13: /The/Original/Name(argExpr1,argExpr2)[sem,3]
e5.14: /The/Rewritten/Name(argExpr1,argExpr2)[sem,3]
```

The two expressions are equivalent according to the aliasing table. If the initial expression $e_{5.13}$ is rewritten to $e_{5.14}$, it is forwarded in direction of `/The/Rewritten/Name`. $e_{5.14}$ in not forwarded in direction of `/The/Original/Name` because one cannot assume that upstream nodes towards `/The/Original/Name` always know about the equivalence, i.e. about rule set '3'. The interest is likely to strand and being NACK'ed with 'missing forwarding rule' as reason. Appending the equivalence information to the interest as 'forwarding hint' in form of the rule set is unfeasible. Rule sets are likely to be (much) large than a single expression/interest because they contain several expressions. Instead, it is possible to forward original and rewritten interests individually and independently of each other, e.g. if the original interest could not be resolved, or to probe all possible paths in parallel.

Forwarding Decisions

In case of nested expressions, it is always the outermost identifier that is relevant for the forwarding decision. Both LPM and exact matching of the relevant identifier against the FIB is applicable. However, argument expressions are not *directly* part of the match. Accordingly, rewriting argument expressions does not influence any forwarding at first. Their rewriting

influences forwarding only *indirectly* in combination with λ-expression rewriting as explained for NFN (see subsection 2.3.2). To give an example, imagine rule set '4' that generically defines rules to rewrite '`argExpr1`' into '`rewrittenArgExpr1`'. A further rule set '5' defines the rules of λ-calculus, i.e. how to detect and manipulate λ-expressions.

```
Rule set '4': argExpr1 → rewrittenArgExpr1
Rule set '5': rules of λ-calculus
```

With the help of rule set '4', expression $e_{5.15}$ can be rewritten to $e_{5.16}$. However, this rewriting only influences forwarding if $e_{5.16}$ is additionally rewritten to the λ-expression $\lambda_{5.1}$, including the according attribution `[sem,5]`. The default inversion mapping of λ-expression $\lambda_{5.1}$ results in name $n_{5.1}$, making clear that the interest is now forwarded in direction of the `rewrittenArgExpr1`.

```
e5.15: /The/Original/Name
       (argExpr1[sem,4],argExpr2)[sem,3]
e5.16: /The/Original/Name
       (rewrittenArgExpr1,argExpr2)[sem,3]
λ5.1:  [(λx.(/The/Original/Name[sem,3] x argExpr2))
       rewrittenArgExpr1][sem,5]
n5.1:  [rewrittenArgExpr1 |
       (λx.(/The/Original/Name[sem,3] x argExpr2))][sem,5]
```

Passing of Argument Expressions

Concerning forwarding, also the passing of argument expressions must be considered. Routine arguments, especially if they are not publicly and permanently available over a routable content name, should be passed by value. This is the most direct way and does not require any additional negotiation to shift arguments to where they are needed.

However, this bears some problems when 1. argument expressions are large and 2. when argument expressions contain sensitive information. In those cases, argument expressions should first be published under a unique routable name, preferably on the requester itself. If the requester is not reachable over a routable prefix, the requester must upload the content to a routable intermediary, e.g. a third-party data depot, and replace the argument expression with the name given by the depot. Making expressions available as normal content objects, no matter if node-local or in a remote data depot, enables to enforce encryption and content-based access control mechanisms, e.g. as described by Yu et al. (82).

For our intents, though, publishing argument expressions under unique names is disadvantageous because it makes it impossible to determine any equivalence between expressions, even if they would be syntactically equivalent.

PIT Timings

PIT timings can also be outlined generally for all classes. As described above in section 5.1, routine calls can be distinguished from static content requests by notation. Routine calls always have a (maybe empty) list of arguments while the list of arguments is absent for static content requests. Static content should be requested rather aggressively. This means that pending interests should be re-triggered rather promptly if not being satisfied with a data packet, i.e. in about one RTT. Assuming a system with NACKs, expired PIT timers for static contents indicate overloaded data sources or that packet losses occurred (interest or data packet), e.g. due to congestion or physical reasons. However,

re-polling should be repeated only a few times in order not to aggravate a potential congestion. Instead, interests should be NACK'ed.

However, the situation is different for routine calls. Computations can take much longer than one RTT. Hence, PIT timers should expire later and even later being NACK'ed. For example, it would be possible to have progressively increasing re-polling intervals for routine calls, e.g. 2 RTTs, 4 RTTs, 8 RTTs. However, this does not solve the problem of truly long running computations. A solution is to use *thunks* as proposed by Sifalakis et al. (38). A thunk is generated from a node that is willing to do the requested computation. It contains a temporary name under which the result will be retrievable once it is available and an optional completion time estimate. Completion time estimates are not only informative for initial requesters to decide when to ask for results, they can also be used to update PIT timers while the thunk travels back to requesters. PITs should not re-trigger interests before their completion time estimates have elapsed. A thunk should not erase PIT entries such that a result, possibly computed faster than estimated, still can travel back to its requesters. Nevertheless, also thunks involve some computations, i.e. choosing a thunk name and computing a completion time estimate. Hence, it is desirable that a node recognizes routine calls a priori and sets a less aggressive PIT timer than for static content, e.g. 1.5-2 RTTs.

5.3 Working with Referentially Transparent Expressions

Referentially transparent expressions always refer to the same result, i.e. they are named unambiguously. CCN/NDN and NFN only allow unambiguous *and* unique names. However, our approach extends in direction of non-unique names. It is allowed that the same result has multiple names. However, these names must be coupled according to some rules. These rules define semantic equivalences between the names. The syntactic 1:1 equivalence is covered by the less constrained semantic N:1 equivalence that does not require unique names. Technically, the syntactic equivalence is a semantic equivalence where no other rule than the syntactic equivalence applies. However, it is conceptually clearer to treat the syntactic equivalence separately because it is the only class that precludes in advance any interpretation of the name. This is an important difference to expressions with one or several meanings. For example, '11' is just a chain of two symbols in its syntactic interpretation. Nevertheless, if the decimal numeral system is taken as a basis, it appears to mean 'eleven', or 'three' if taking the binary numeral system as basis.

Accordingly, the relevant classes in the space of referentially transparent functions are syntactic (`[syn]`) and semantic (`[sem, ruleSetId]`). Their advantage is that their results are always cacheable because they do not have side effects and their return values do not vary. This section focuses on interest aggregation and content matching for these two classes.

5.3.1 Interest Aggregation

For `[syn]` and `[sem,X]` expressions, the question is not whether interests should be aggregated or not, only how. Interest aggregation is interest-to-interest matching, i.e. the matching of an inbound interest on its upstream path towards the source against the list of interests stored in pending interest tables

(PITs). What is matched against each other are the expressions within the interests. Other elements of an interest, e.g. the nonce, are not involved in the matching process. Nevertheless, nonce values are still needed for the detection of duplicate interests. For the examples, we are assuming no duplicates. A positive match, i.e. when an equivalent interest was found in the PIT, means that the inbound interest is appended to the equivalent PIT entry without being forwarded. If no equivalent PIT entry was found, a new PIT entry is created, and the interest is forwarded as usual.

Aggregation of [syn] Expressions

Interests that carry expressions with the `[syn]` attribution are aggregated like CCN/NDN interests, i.e. they aggregate with syntactically equivalent PIT entries (83), (2). For example, imagine having a node N_1 with a PIT table such as in Table 5.2. Initially, the PIT table contains only entries $p_{5.1}$-$p_{5.3}$, without the arrival interface in parentheses.

Table 5.2 – PIT of Node N_1: Syntactic Examples

Entries below the red dashed line and in parentheses are not part of the initial state. They are added during the example.

Entry	Content Name	Arrival Interfaces
$p_{5.1}$	`/Static/Content/ABC/v1/c1[syn]`	1
$p_{5.2}$	`/Static/Content/ABC/v1/c2[syn]`	1 (,4)
$p_{5.3}$	`/Function/Call/Add(2,3)[syn]`	3
$p_{5.4}$	`/Function/Call/Add(3,2)[syn]`	3
$p_{5.5}$	`/Function/Call/Add(5,0)[syn]`	3
$p_{5.6}$	`/Function/Call/Add(1+1,3)[syn]`	3

If an interest $i_{5.1}$ arrives on interface 1 at node N_1, it is aggregated with PIT entry $p_{5.2}$. The list of arrival interfaces for entry $p_{5.2}$ remains unchanged, even if the new interest $i_{5.1}$ has a different nonce than the interest that was responsible for the creation of PIT entry $p_{5.2}$. If another interest $i_{5.2}$ for the same content arrives over interface 4, $p_{5.2}$ is updated by appending the interface identifier to the list of arrival interfaces.

```
i5.1: /Static/Content/ABC/v1/c2[syn] over interface 1
i5.2: /Static/Content/ABC/v1/c2[syn] over interface 4
```

PIT entry $p_{5.3}$ implies that node N_1 does not possess the function `/Function/Call/Add(argExpr1,argExpr2)`. Otherwise, the node would have satisfied the request and the entry would never have appeared in the PIT. Assume that the function is doing what its name suggests: it takes two numbers as input and returns their sum. While interest $i_{5.3}$ aggregates with $p_{5.3}$, interests $i_{5.4}$, $i_{5.5}$, and $i_{5.6}$ do not aggregate with $p_{5.3}$, although they will have the exact same result. Interests $i_{5.4}$-$i_{5.6}$ induce a new PIT entry each, i.e. $p_{5.4}$-$p_{5.6}$. The aggregation decision is always independent of the arrival interface. However, for the sake of completeness, we assume that all four interests $i_{5.3}$-$i_{5.6}$ arrived on interface 3.

```
i5.3: /Function/Call/Add(2,3)[syn]      over interface 3
i5.4: /Function/Call/Add(3,2)[syn]      over interface 3
i5.5: /Function/Call/Add(5,0)[syn]      over interface 3
i5.6: /Function/Call/Add(1+1,3)[syn]    over interface 3
```

Aggregation of [sem,X] Expressions

The aggregation of interests with `[sem,X]` expressions involves a more complex lookup process. A node that was not able to satisfy an interest should next try to match the *unreduced* interest against the PIT. A match implies that the node is not able (or willing) to reduce the expression and that an equivalent request has already been forwarded. Only if there is no match, a node should check if it possesses rule set X. If it possesses the rule set, it can apply it and check the *reduced* interest against the CS and the PIT. There may be matches now.

Applying rule sets is not mandatory yet recommended. An overloaded node can jump over this step, even if it possesses the necessary rule set. Moreover, a node can completely omit interest aggregation and just forward any interest. However, it remains an open question how helpful this is to reduce the workload of a node. According to Dabirmoghaddam et al. (83), normal, i.e. syntactic, interest aggregation may anyway be vain endeavor. Fallback names and function aliases should only be used in cases where the retrieval was unsuccessful. Otherwise, rewriting is unnecessary work, although not harmful.

Coming back to the positive case of anonymous expression blocks where the node possesses the rule set(s) and has free resources to apply them, it should do it. There are three reasons why. 1. Applying calculi-based equivalence rules corresponds to performing computations on the name. This kind of distributed name resolving is just as well part of edge and fog computing as the shifting of routine declarations itself. Likewise, it reduces the computational load of function sources. 2. Applying rules as early as possible increases chances for interest aggregation. 3. Furthermore, applying rules as early as possible also increases the probability to match cached results. This entails the avoidance of unnecessary computations, economized transmissions and lower latencies.

For example, imagine another node N_2 with only one initial entry ($p_{5.7}$, Table 5.3) in its PIT. Moreover, assume that node N_2 possesses rule set 1, i.e. the rules to detect and interpret mathematical expressions. In addition, assume that node N_2 does not possess the implementation of `/Function/Call/Add(argExpr1,argExpr2)` such that interests for that function are not resolved, but added to the PIT and forwarded.

Table 5.3 – PIT of Node N_2: Semantic Examples

Initially, the table only contains entry $p_{5.7}$, without equivalence information. All entries below the dashed red line are added during the example.

Entry	Content Name	Arrival Interfaces
$p_{5.7}$	`/Function/Call/Add(` `[2][syn],[3][syn])[syn]`	3
	↓ Equivalence Information ↓	
	`/Function/Call/Add(` `[1+1][sem,1],[3][syn])[syn]`	1
	`/Function/Call/Add(` `[1+1][sem,1],[1*3][sem,1])[syn]`	1
$p_{5.8}$	`/Function/Call/Add(` `[3][sem,1],[2][sem,1])[syn]`	4
$p_{5.9}$	`/Function/Call/Add(` `[5][sem,1],[0][sem,1])[syn]`	4
$p_{5.10}$	`/Function/Call/Add(1+1,3)[syn]`	2

Both interests $i_{5.7}$ and $i_{5.8}$ aggregate with PIT entry $p_{5.7}$ because both interests reduce to the same expression that is equivalent to the expression already present in the PIT. According to the generalized CCN/NDN execution model (see subsection 2.2.2), the new arrival interfaces would simply be added to PIT entry $p_{5.7}$. However, this is not enough for our purpose. The equivalence information cannot be discarded because node N_2 must distribute incoming data packets back on different requests. Hence, in order that the equivalence information is not lost, *unreduced* expressions are appended to $p_{5.7}$. Note that in those cases where an expression is reduced but does not match an existing PIT entry, only the *reduced* interest is forwarded. Its arrival interface is empty, and the original interest is in the list of equivalent interests, including the original arrival interface.

In contrast, $i_{5.9}$ and $i_{5.10}$ will again not aggregate with $p_{5.7}$ because they reduce to different expressions, even though their final results will be the same (=5). Hence, they are separately added to the PIT (entries $p_{5.8}$ and $p_{5.9}$) and forwarded independently.

```
i5.7: /Function/Call/Add([1+1][sem,1],[3][syn])[syn]
      over interface 1
i5.8: /Function/Call/Add([1+1][sem,1],[1*3][sem,1])[syn]
      over interface 1
i5.9: /Function/Call/Add([3][sem,1],[2][sem,1])[syn]
      over interface 4
i5.10: /Function/Call/Add([5][sem,1],[0][sem,1])[syn]
      over interface 4
```

A further interesting request is interest $i_{5.11}$. As outlined above, not attributed argument expressions without routine calls can be checked for syntactic equivalence. Thus, the first argument of $i_{5.11}$, '`1+1`', is the same as `[1+1][syn]`. The same applies to the second argument expression.

```
i5.11: /Function/Call/Add(1+1,3)[syn] over interface 2
   =  /Function/Call/Add([1+1][syn],[3][syn])[syn]
```

Determination of Equivalence

The question is if $i_{5.11}$ should aggregate with $p_{5.7}$, or more concretely with $i_{5.7}$ that has been stored as equivalent to $p_{5.7}$. The short answer is no, because $i_{5.7}$ and $i_{5.11}$ are not syntactically equivalent. The explanation: Outside the function `/Function/Call/Add`, it *can* be determined that `[1+1][sem,1]` equals `[2][sem,1]`, which is equivalent to `[2][syn]`. However, it *cannot* be determined that `[1+1][syn]` equals `[2][syn]`. Only if the function has the same interpretation of '`1+1`' as rule set 1, `[1+1][syn]` and `[1+1][sem,1]` lead to the same result. However, this cannot be assessed from outside the function. Hence, PIT entry $p_{5.10}$ is added and $i_{5.11}$ must be forwarded individually. With the same line of arguments, it can be shown that, *without* knowing rule sets X and Y, it is impossible to determine an equivalence between $e_{5.17}$ and $e_{5.18}$, even if the given expressions are syntactically equivalent apart from their attributes.

```
e5.17: <arbitrary expression>[sem,X], e.g. [1+1][sem,1]
e5.18: <arbitrary expression>[sem,Y], e.g. [1+1][sem,2]
```

Nevertheless, note that due to our recommendation to only determine equivalences between fully qualified identifiers, two *syntactically* equivalent

routine calls are always equivalent, even if their identifiers are subject to two different aliasing tables. This is, expression $e_{5.19}$ must be equivalent to expression $e_{5.20}$, given that their argument expressions are equivalent (see Figure 4.4, page 84).

```
e5.19: /Name/Of/Function(argExpr1)[sem,X]
e5.20: /Name/Of/Function(argExpr1)[sem,Y]
```

Table 5.4 summarizes the proceeding concerning interest aggregation for referentially transparent expressions.

Table 5.4 – Interest Aggregation for RT Expressions

[syn]	→	Yes, syntactical over the entire expression.
[sem,X]	→	Yes, semantical over the fully qualified identifier or the delimited application sphere according to rule set X. Parts of the expression without explicit attribution must match syntactically.

5.3.2 Caching & Content Matching

Due to the design decision to store equivalence information in the PIT, redistributing incoming data packets back to arrival interfaces is rather simple. The content name of an incoming data packet is matched against the *top entries* of all PIT entries. Only entries at the top have been forwarded, while equivalent entries have been aggregated. Hence, only data packets with names of such top entries should be received. If there is no PIT match, data packets are dropped. If there is a match, the intermediate node is responsible to *re-pack* the content from the incoming data packet with the names from the equivalence list. Thereafter, the intermediate node must *re-sign* the packet and send it downstream according to the list of arrival interfaces. The packet flow remains symmetric.

Re-packing & Name Binding

Re-packing creates a new data packet with the payload of the original data packet and involves two steps. First, the content name of the original data packet is extended with all equivalent names. Hence, the new packet has a list of names instead of a single name. Equivalences must not be re-determined. They can be taken from the matching PIT entry. Second, the re-packing node must populate the signed info section with its information (hash algorithm, key locator, etc.) and add a newly computed signature. For referentially transparent results with unambiguous names, this approach is feasible because results are valid permanently.

Transitive Trust

Through its signature, the re-packing node confirms the equivalence it determined. A requester that trusts a result signed by a re-packing node, transitively trusts other sources of partial results. For example, if a data packet, originally signed by node C, is re-packed and re-sign by an intermediate node B, receiving node A trusts node B insofar that B itself is trustworthy, as well as that node B only accepts data packets from other trustworthy sources. Hence, A trusts that node B verifies the authenticity and integrity of data packets before re-signing them. Nevertheless, node B should cache the original data packet from node C such that the original signature can be verified if desired.

So far, equivalence information was only stored in transient PIT entries. By adding equivalence information into data packets, the information is permanently made available. However, to make this information detectable, it should be reflected in the content store. Therefore, CS entries should be extended by a list of equivalent content names, too. This can be implemented rather memory-efficient because equivalent names need only to point to one common data packet. In best N:1 manner, there is no need to cache data packets redundantly. Table 5.5 shows how the CS of node N_2 from above looks like after having received all data packets according to PIT entries of Table 5.3.

Table 5.5 – CS for Node N_2: Data Pointers

Whenever interests have been aggregated due to an equivalence, the resulting data packet needs to be re-packed. This includes appending equivalent names to the data packet and its re-signing. The payload remains unchanged. The re-packed data packet $dp_{5.7}$ should not be cached redundantly for every equivalent name. Pointers (dotted arrows) to the packet are enough. Finally, the re-packed data packet can be relayed back to requesters.

dp_x = data packet that has satisfied PIT entry p_x and corresponding to CS entry cs_x.

Entry	Content Name	Data Packet
$cs_{5.7}$	$n_{5.7.1}$: /Function/Call/Add ([2][syn],[3][syn])[syn]	$dp_{5.7}$[{$n_{5.7.1}$, $n_{5.7.2}$, $n_{5.7.3}$}, $signature_{new}$, signed $info_{new}$, $payload_{original}$]
	↓ Equivalence Information ↓	
	$n_{5.7.2}$: /Function/Call/Add ([1+1][sem,1],[3][syn]) [syn]	
	$n_{5.7.3}$: /Function/Call/Add ([1+1][sem,1],[1*3] [sem,1])[syn]	
$cs_{5.8}$	$n_{5.8}$: /Function/Call/Add ([3][sem,1],[2][sem,1]) [syn]	$dp_{5.8}$[$n_{5.8}$, signature, signed info, payload]
$cs_{5.9}$	$n_{5.9}$: /Function/Call/Add ([5][sem,1],[0][sem,1]) [syn]	$dp_{5.9}$[$n_{5.9}$, signature, signed info, payload]
$cs_{5.10}$	$n_{5.10}$: /Function/Call/Add (1+1,3)[syn]	$dp_{5.10}$[$n_{5.10}$, signature, signed info, payload]

Adaptions

Both protocol and architecture need small adaptions to comply with the proposed approach. Unreduced expressions that could not be satisfied by the content store and not yet matched a PIT entry are, whenever possible, reduced as far as possible by applying all locally available rewriting rules. Whenever a rewriting was applied, the node tries to match the rewritten expression against the CS, and if there is still no match, also searches for matching PIT entries. In case of a PIT match, the original, non-rewritten expression is aggregated with the match. In case of no PIT match, both rewritten and original expressions are added to the PIT. However, only the rewritten interest is forwarded. The arrival interface of the forwarded rewritten interest is empty, while the aggregated original interest is associated with its arrival interface.

The payload of an incoming data packet, if matching a PIT entry, is equipped with the equivalent names of the matching PIT entry. The new re-packed data packet must be newly signed. Both happens before the data packet is relayed back to requesters. This extended data packet has an increased probability to satisfy upcoming interests. As described above, PIT entries with empty arrival interface information may appear. They are ignored in the data return process.

On the architectural side, no new data structure is needed. The PIT must be adapted such that it maps a list of equivalent names (instead of only a single name) to a list of arrival interfaces. Similarly, the CS must be adapted to map a list of equivalent names to a data packet. Finally, also data packets must support list of names.

Preservation of Equivalence Information

As the equivalence information is stored in the CS, the information is lost whenever content is evicted from the cache. Equivalences can be re-determined. However, before removing entries form the CS, it is also possible to preserve the equivalence information by writing it to a new aliasing table with a new rule set identifier. The aliasing table can be stored again in the CS, possibly with a remark not to evict it too early.

Table 5.6 – Caching and Content Matching for RT Expressions

[syn]	→	Yes, syntactical over the entire expression.
[sem,X]		Yes, semantical over the fully qualified identifier or the delimited application sphere according to rule set X. Parts of the expression without explicit attribution must match syntactically.

5.4 Working with Referentially Opaque Expressions

The previous section covered cases with multiple names for the same content. This section targets referential opacity, i.e. cases with multiple results for the same name.

Referentially opaque routines have the advantage of being able to manipulate state outside of their scope. This is the main reason why they should be supported. However, another advantage of using them is the hassle-free implementation of many useful applications with continuously changing contents. As discussed in subsection 3.3.2, it is possible to implement such scenarios with referentially transparent names, too. However, this is cumbersome because name space design and request for latest content problems must be solved instead. In contrast, referentially opaque names must be published just once. There is no need to publish and coordinate "versioned" names. Relevant equivalence classes in the space of referentially opaque routines are ephemeral `[eph]`, commutative `[com]` sequence-idempotent `[seq]` and problematic `[prb]`. As done for referentially transparent names, these classes will be investigated on their ability for interest aggregation, permanent caching and content matching properties.

5.4.1 Interest Aggregation

In section 4.3 on adaptive equivalences, four expression classes have been investigated towards how the determination of an equivalence should be adapted when it comes to interest-to-interest matching, i.e. upstream interest aggregation in the PIT. We elaborated the following four proceedings concerning interest aggregation:

- Ephemeral expressions → no (default), opt-in
- Commutative expressions → no
- Sequence-idempotent expressions → yes
- Problematic expressions → no

The circumvention of interest aggregation or content store matching is a non-issue in CCN/NDN. In our case, for functional correctness beyond referential transparency, the issue needs to be addressed. Avoiding the aggregation of interests is simple when making use of ECS tags. The logic can simply be built in the nodes. Interests that carry an `[eph,aggr]` or `[seq]` expression must be aggregated, interests carrying either an `[eph]`, `[com]` or `[prb]` expression must not be aggregated. For example, a Unix timestamp request, attributed with `[eph,aggr]` like in $i_{5.12}$, travels towards the function provider but aggregates with the first syntactically equivalent pending interest on its way. In contrast, interest $i_{5.13}$ never aggregates. Another example that does not aggregate is $i_{5.14}$ that carries a commutative operation.

```
i5.12: /Service/Provider/Name/CurrentUnixTime()[eph,aggr]
i5.13: /Service/Provider/Name/CurrentUnixTime()[eph]
i5.14: /Name/Of/CmRDT/Replica/1/Increment()[com]
```

Aggregation of interests, whenever allowed, can be performed directly by name. There is no need for an additional discriminative element outside the name. In contrast, there must be a way to discriminate homonymous interests that do not aggregate. Otherwise, different but also homonymous result data packets cannot be re-assigned to corresponding interests.

The only proposition how to influence interest aggregation was in connection with the retrieval of latest information by Shi (75), as discussed in subsection 3.3.1. The discussed approach proposed to append a random number, i.e. an additional random name component. This approach works for both CCN and NDN in all cases where data packets are created dynamically (~on-demand). For example, interests $i_{5.15}$, $i_{5.16}$, and $i_{5.17}$ can be used instead of $i_{5.14}$. They do not aggregate. Unlike the problems that occur with previously created content, i.e. where requesters have either to guess the discriminative elements, retrieve it in a previous request (see subsection 3.3.3) or to use the non-deterministic 'CanBePrefix' selector (see subsection 3.3.2), there are no such problems with dynamically created responses.

```
i5.15: /Name/Of/CmRDT/Replica/1/Increment()[com]/po9idn3e
i5.16: /Name/Of/CmRDT/Replica/1/Increment()[com]/Hi8M-3Dx
i5.17: /Name/Of/CmRDT/Replica/1/Increment()[com]/1kG834TT
```

This approach does only work if the random name component is treated as every other preceding name component and matched exactly against the PIT. In contrast, the provider of the increment function must be able to distinguish the random name component from the rest of the expression. Otherwise, the provider cannot differentiate between referentially opaque function calls and requests for content objects. Fortunately, this is easy to achieve on the packet format level, e.g. when expressed in the type-length-value (TLV) scheme.

However, at least as far as NDN is concerned, the random name component is redundant and therefore a waste of transmission capacity. NDN interests already carry a *discriminative element* in addition to (and outside of) the name, i.e. the nonce. If the nonce is globally unique within a time frame long enough to detect duplicate interests, it is also sufficiently unique to discriminate two interests for the same function call within a narrow time frame. Instead of $i_{5.15}$, $i_{5.16}$, and $i_{5.17}$, one can use $i_{5.18}$, $i_{5.19}$, and $i_{5.20}$. They have the same distinctiveness. '`n_`' denotes the nonce value, that itself is given as hex value, indicated by the leading '`0x`'. Braces shall indicate that nonce values are not exposed in names. They are contained in interests.

```
i5.18: /Name/Of/CmRDT/Replica/1/Increment()[com]
       {n_0xf3b029c1}
i5.19: /Name/Of/CmRDT/Replica/1/Increment()[com]
       {n_0x4a18337e}
i5.20: /Name/Of/CmRDT/Replica/1/Increment()[com]
       {n_0xc2468acf}
```

Adaptions

In NDN, the nonce is a 32 bit long string that "*should uniquely identify an Interest packet*" (84). Likewise, we argue that this is sufficiently unique[36] to use it for the discrimination of not aggregated interests. NDN interest packets therefore fulfil all requirements. CCN interest packets, however, do no longer carry a nonce since the change from version 0.x to 1.x. The given reason for this transformation is that "*Nonces breaks (sic) aggregation*" (23) and "*Interests can't be aggregated*" (23). Because this is exactly the desired capability, CCN interest packets would have to re-introduce nonce values.

On the protocol side, the workflow needs some adaption. Whenever an interest must not be aggregated, i.e. for `[eph]`, `[com]` and `[prb]` expressions, the step of searching PIT matches can be jumped over. If not being satisfied, an interest can immediately be added to the PIT and forwarded. Important is that the nonce is stored along with the name in the PIT entry. Otherwise, it is not possible to discriminate the entries. Note that duplicate interests have been filtered out and discarded previously by checking their nonce values against the recently seen nonce values list (see subsection 2.2.1). Hence, there is no risk that a `[com]` or `[prb]` expression is evaluated twice for the very same request. Whenever an interest shall be aggregated, i.e. for `[eph,aggr]` and `[seq]` expressions, the PIT must be search for matches as usual. That means, compared expressions must match syntactically, including all arguments, but without the nonce. Only in case of no match, the interest will be forwarded. Like for `[syn]` and `[sem,X]` expressions, the aggregation of `[eph,aggr]` and `[seq]` expressions does not require to store nonce values along with names in the PIT. Names alone are discriminative enough. Table 5.7 lists the theoretical procedure of interest aggregation for adaptive equivalence classes.

[36] A deeper discussion about this issue follows in section 8.1.

Table 5.7 – Adaptive Interest Aggregation for RO Expressions

[eph]	→	No.
[eph, aggr]	→	Yes, syntactical over the entire expression. The nonce is irrelevant for aggregation.
[com]	→	No.
[seq]	→	Yes, syntactical over the entire expression. The nonce is irrelevant for aggregation.
[prb]	→	No.

The way how we discriminate interests from each other also influences the way how interests are matched against data packets. It does not matter if *interests are matched against cached contents upstream*, or if *data packets are matched against PIT entries downstream*. Both is discussed in the following subsection.

5.4.2 Adaptive Content Matching & Caching

Among referentially opaque expressions, only sequence-idempotent expressions (`[seq]`) have names that are specific enough to re-assign result packets to requests. It does not matter if result packets originate from a cache or an upstream node. Hence, only the syntactic equivalence over the expression should be determined. The nonce must not be part of the match because nonce values change from call to call, even if two calls invoke the same routine with the same arguments. Considering the nonce in the matching process would therefore lead to a second evaluation of a sequence-idempotent expression. This is not harmful but also not desirable.

`[eph]`, `[com]` and `[prb]`expressions should not be satisfied with result packets that only have a matching name. Such requests are always unique and so must be their results. To disambiguate homonymous interests for individual PIT entries, we proposed to use sufficiently unique nonce values. These nonce values must be reused to disambiguate request-specific result packets and to uniquely re-assign them to requests. To do so, nonce values of requests must be replicated in responses. This solution works because responses are (obliged to be) produced on demand. Consequently, nonce values are known on packet production time. There is no need to guess nonce values.

Exceptions are `[eph,aggr]` expressions because they are matched to content whether by name only nor request-specifically by name and nonce. Nonce values are too specific because a response should satisfy more than one request, namely all aggregated requests. Other matching criteria need to be considered instead. For example, subsection 4.3.1 suggested to use a monotonically increasing timestamp that enables to check whether a response is recent enough or not. In combination with a timestamp, further matching criteria are thinkable. For example, producers can add GPS coordinates to responses. They only match if they were produced within a certain perimeter (~geofencing). As soon as the producer leaves the area, responses no longer match requests. However, note that such indistinct matching criteria often entail that several different result packets match a request. Accordingly, a requester must comply with one of them, given that it fulfills its specification. In fact, this is the non-deterministic retrieval scheme of NDN we criticized in section 3.3. Nevertheless, the behavior is now reflected by the `[eph,aggr]` attribute and therefore visible for everyone. Table 5.8 summarizes adaptive content matching for referentially opaque expressions.

Table 5.8 – Adaptive Content Matching for RO Expressions

[eph]	→	Syntactical over the entire expression and nonce.
[eph,aggr]	→	Syntactical over the entire expression and further matching criteria but not over the nonce.
[com]	→	Syntactical over the entire expression and nonce.
[seq]	→	Syntactical over the entire expression.
[prb]	→	Syntactical over the entire expression and nonce.

It seems that request-specific results can immediately be dropped after having satisfied corresponding PIT entries. However, it will turn out for technical reasons that caching them shortly is the better advice than overhasty dropping. Accordingly, we distinguish permanent caching (5.4.2.1) from short-term caching (5.4.2.2). This differentiation helps to improve garbage collection, one of the soft spots of NFN (see 3.1). Moreover, two examples address how to work with expressions that intrinsically do not have return values.

5.4.2.1 Permanent Caching & pACKs

In the space of referentially opaque expressions, only results of sequence-idempotent expressions are worth to be considered for *permanent* caching. The reason is that they have a unique one-time side effect that remains stable over a sequence of operations. Hence, whether a repeated invocation nor their position in the sequence influence the result. Although a repeated invocation would not cause any problems in terms of consistency, there are two players who should have interest to avoid waste work. 1. The owner of a manipulated state can save a repeated invocation of a call, and 2. all intermediate nodes and internet service providers between requester and function provider can save bandwidth and unnecessary forwarding efforts. Owners of modifiable states can decide whether (and how excessive) they prefer to keep track of handled expressions or if they rather prefer to occasionally re-evaluate some already handled expressions. The former implies additional memory complexity, the latter additional time complexity.

Permanent Acknowledgment

Internet service providers can relieve their communication infrastructure by caching return values of sequence-idempotent expressions. However, the main purpose of sequence-idempotent expressions is their side effect. Hence, there must not necessarily be a return value. If the evaluation does not create any return value, there is virtually nothing to cache. Nevertheless, some sort of generic return value must anyway be sent back to the requester to 1. inform the requester about the reception and successful evaluation of her/his request and 2. to clear all PIT entries on the path. Accordingly, a generic return value can be understood as acknowledgment and must only carry the name of the interest. However, instead of discarding the acknowledgment after clearing PIT entries and sending it further downstream, intermediate nodes should cache it. An empty data packet serves as good as any other return value to satisfy later interests. We call these empty result packets *permanent acknowledgments*, i.e. *pACKs*.

Example

The merge function of a state-based CvRDT counter can serve as an example. The CvRDT counter implementation takes the approach to reflect the total counter reading as vector of replica-individual counters. The total counter reading is the vector sum. Each replica holds a copy of the vector, i.e. the

CvRDT. Whenever a specific replica receives an increment, it increments its own counter. However, instead of sending the update operation to all other replicas, the updated vector itself is transmitted. They then merge their vectors with the received vector by taking the maximum (~supremum) of each element. Increments fulfill the requirement to only monotonically increase the internal state, and the supremum function meets the requirements of a lattice, as shown in subsection 2.5.1.2.

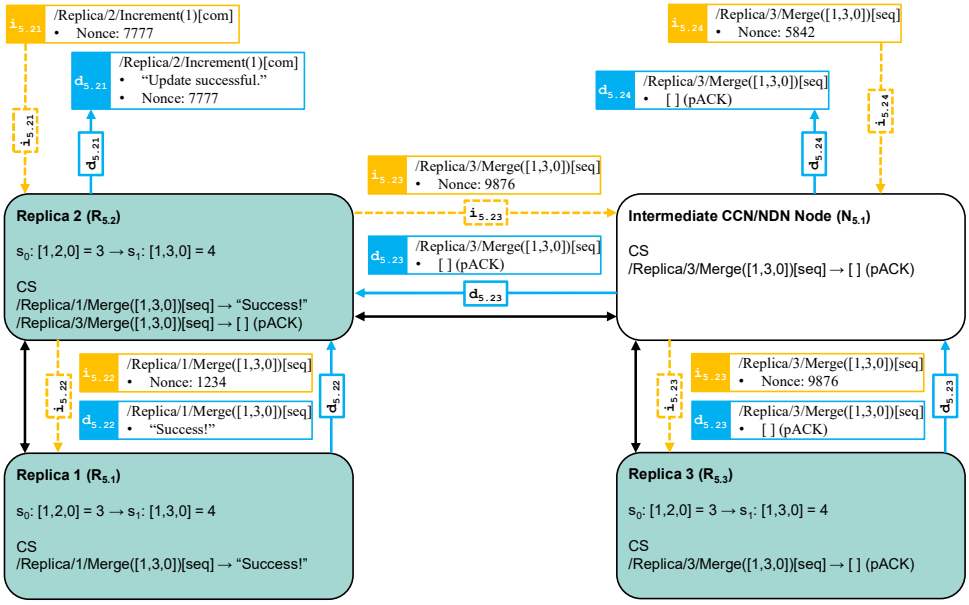


Figure 5.2 – Caching of Sequence-Idempotent Expressions

This figure shows a scenario with three replicas ($R_{5.1}$, $R_{5.2}$, $R_{5.3}$) and one normal CCN/NDN node ($N_{5.1}$). Initially, all content stores (CS) are empty and each replica stores the up-to-date replicated state s_0, i.e. the vector of node-individual counters. The total counter reading is sum([1,2,0]) = 3 for all replicas. Gradually, content stores get populated with data packets (explicit result or pACK) that are returned over the nodes. The figure shows final CS states, after all data packets have been sent back to their requesters.

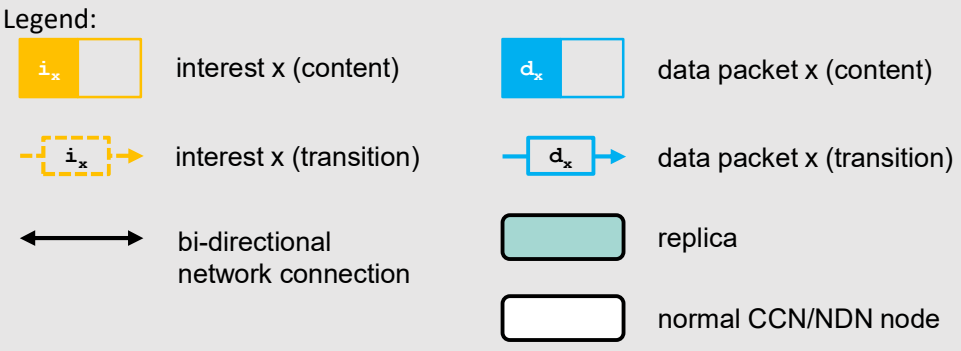


Figure 5.2 details the update and merger sequence after an initial increment $i_{5.21}$ of replicated counter s_0 at replica $R_{5.2}$, including how to profit from pACKs. A requester can increment the replicated counter by calling the increment function of one replica, e.g. the one of replica $R_{5.2}$ by issuing $i_{5.21}$. $R_{5.2}$ updates its internal state to s_1, which now sums up to 4, and informs the initial requester

with $d_{5.21}$ about the successful update. Subsequently, $R_{5.2}$ send its updated state s_1 to replicas $R_{5.1}$ and $R_{5.3}$ for merger by issuing $i_{5.22}$ and $i_{5.23}$ *concurrently*. After merger, both replicas should answer with either an explicit return value (e.g. "Success!" as in $d_{5.22}$) or with a permanent acknowledgment ([] (pACK) as in $d_{5.23}$). Empty brackets indicate the empty content of pACKs. Both cases enable replica $R_{5.2}$ to detect success or failure of its merger requests. If problems occur, replica $R_{5.2}$ can intervene, e.g. by re-sending failed requests. Normal CCN/NDN nodes as $N_{5.1}$ can cache pACKs like every other data packet. Besides that, pACKs will satisfy all later interests that carry an already merged state, even if they have other nonce values than their syntactically equivalent predecessor. For example, if $i_{5.23}$ was already handled successfully, $i_{5.24}$ will be intercepted and satisfied by the pACK in the CS of CCN/NDN node $N_{5.1}$.

Adaptions

No architectural adaption is required to handle sequence-idempotent results. pACKs and any other permanent return value can be added to content stores the same way as result packets of referentially transparent expressions.

5.4.2.2 Short-Term Caching & tACKs

For `[eph]`, `[eph,aggr]`, `[com]`, and `[prb]` expressions, the caching policy is 'do not cache permanently'. However, this advice bases on the assumption of a non-faulting network where no data packet is ever lost on its downstream path towards requesters. In practice, however, reliable broadcasting must be ensured. To do so, it is necessary to cache result packets of `[eph]`, `[eph,aggr]`, `[com]`, and `[prb]` expressions for a short time, too. In case of a lost data packet, a downstream PIT timer will expire and re-trigger an identical interest.

Transient Acknowledg-ment

Optimally, such a re-triggered interest gets satisfied by a temporarily cached result in order to economize bandwidth consumption and to shorten response times. This is the first reason why commutative and problematic expressions without a return value should be answered with an empty data packet, too. The second reason is the same as for pACKs, i.e. to notify initial requesters about the success of their operations. However, in contrast to pACKs, empty data packets should only be cached transiently. We therefore call them *transient acknowledgments*, i.e. tACKs. The reason why we do not call them ephemeral acknowledgments is that they can only be emitted by commutative and problematic expression, but not by ephemeral expressions because they always have a non-empty return value.

Example

Figure 5.3 illustrates a transmission scenario with a client $C_{5.1}$ that tries to increment a CmRDT counter on replica $R_{5.4}$ twice by issuing $i_{5.25}$ and $i_{5.26}$. The requests commute, thus, no response must be waited, and the order of evaluation as well as the arrival order of $d_{5.25}$ and $d_{5.26}$ at requester $C_{5.1}$ do not matter. The first increment request $i_{5.25}$ makes it through to replica $R_{5.4}$ where it is locally applied. Replica $R_{5.4}$ answers with an ephemeral return value (data packet $d_{5.25}$), i.e. one that is a unique response to the request. Therefore, it should not be cached to satisfy later equivalent interests. Albeit, intermediary $I_{5.1}$ is well advised to cache it for a short time. Assuming that $d_{5.25}$ is lost somewhere downstream, a re-triggered interest $i_{5.25}$ will appear after a timeout. If intermediary $I_{5.1}$ cached $d_{5.25}$ long enough, it can now intercept and satisfy the re-triggered interest. The second request $i_{5.26}$ made it through both directions without troubles. Its response $d_{5.26}$ arrives at client $C_{5.1}$ before $d_{5.25}$. As mentioned above, this does not cause problems.

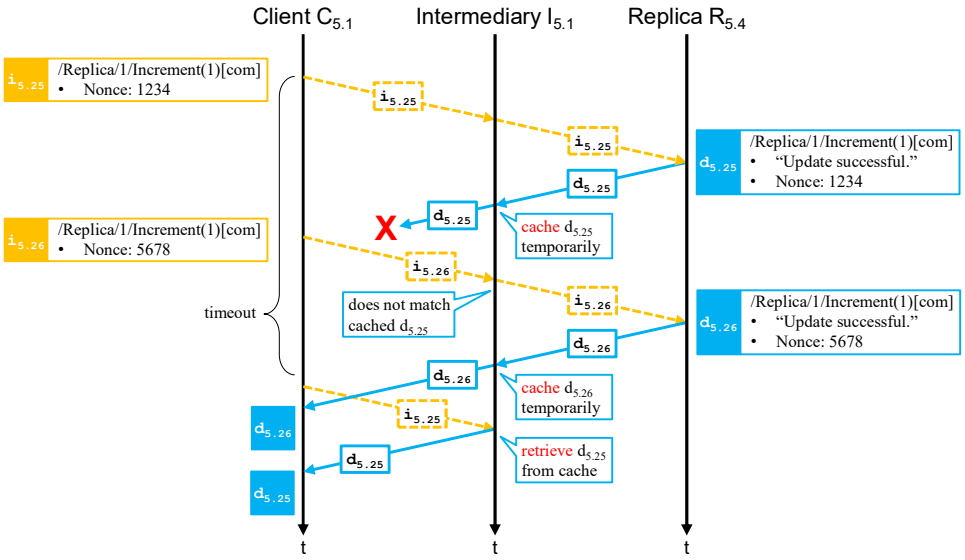
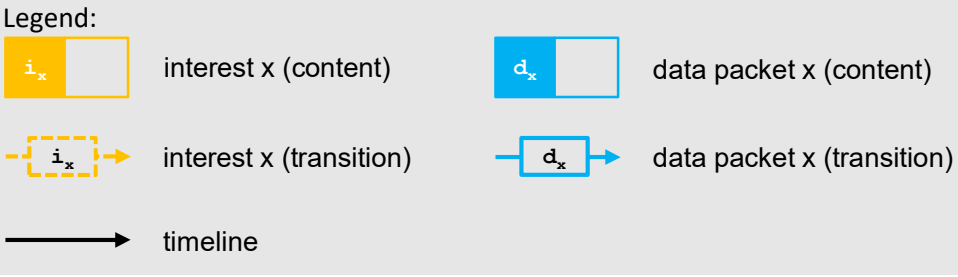


Figure 5.3 – Short-Term Caching of Ephemeral Return Values

To mitigate consequences of packet loss, intermediate nodes should cache ephemeral return values temporarily. Intercepting re-triggered interests helps to save network resources. However, temporarily cached return values must not be used to satisfy upcoming, new interests. This is the reason why every data packet carries the nonce of the interest that caused its creation. Compare $d_{5.25}$ and $d_{5.26}$ from this figure and $d_{5.21}$ from Figure 5.2. Both are responses to commutative expressions and therefore contain a nonce value. In contrast, $d_{5.22}$, $d_{5.23}$, and $d_{5.24}$ from Figure 5.2 are responses to sequence-idempotent expressions and therefore do not need to carry a nonce value. Transient acknowledgments (tACKs) that should be used if – in opposition to this example – no explicit return value is created must contain a nonce value, whereas pACKs get by without nonce values.



Caching Periods

The obvious question is how long opaque results should be cached to intercept as many re-triggered interests as possible, without wasting too much memory due to too permissive caching periods. A simple answer to this question may not exist because several factors influence it. For example, the latency of a specific path influences the answer, as well as the length of the timeout period before an interest is re-triggered. Moreover, it matters if re-triggering is coordinated by initial requesters only, or also by intermediate nodes. However, assuming that only initial requesters re-trigger interests, a rough upper bound estimation can be given: At any intermediate node, a re-triggered interest does not have to be assumed later than:

$$CP_{min} = TP + \frac{1}{2}RTT$$

with CP_{min} the worst-case minimum caching period, TP the timeout period before an interest is re-triggered, and RTT the round-trip time between consumer and producer of the result. This estimation is rough because the network weather can change during transmission and timeout phases. For example, the RTT is not a stable value. However, an interest can collect both information in real-time while traveling from a requester towards a data provider. The requester's timeout period can be appended to the interest at its creation and the travelling time between requester and provider, that is approximately ½ RTT, can be measured. Therefore, the worst-case minimum caching period estimation is available when the data packet is released. Hence, it can be appended to the data packet and read out later from every intermediate node. With this rough approach, nodes closer to an initial requester will tend to cache opaque results rather too long, while nodes close to the source will tend to cache rather too short. Nevertheless, if memory is not extremely constraint, ephemeral return values should rather be cached "a little be too long" than "too short". Caching them too long bears the risk of colliding nonce values from two interests that are meant to be different. However, as the combination of name and nonce matters, the risk is controllable.

Snapshots and Log Compaction

If the estimation was too optimistic, and a re-triggered interest appears after the result has been dropped out of caches, a duplicate evaluation must be prevented, at least in those cases where it matters, i.e. for commutative (`[com]`) and problematic (`[prb]`) expressions. On-path caches can only help to mitigate the problem. Ultimately, no one else than replicas themselves can take this duty. They must keep record of what happened so far. For example, CRDTs are often implemented to keep record of all updates in a log. From time to time, when the log has grown to a certain threshold, replicas agree on making a *snapshot*, i.e. they agree on a consistent state. The history of updates can then be forgotten. Related to CRDTs, this process is referred to as *log compaction*.

Adaptions

The implementation of temporarily cached data packets is rather a content store management extension. There is no need for a separate data structure. Content store management is normally limited to the clearance of unpopular content. Classic CCN/NDN does not have to worry about temporal aspects because data packets never lose their validity. For ephemeral return values, the content store should remember the *arrival time* of every data packet. The *age* of entries can then be compared to the worst-case minimum caching period, or any other individual limit. All entries older than the limit can be evicted regularly from the CS. The equation below summarizes the procedure.

$$t_{now} - t_{arrival} = age \begin{cases} age \leq CP_{min} & keep\ data\ packet \\ age > CP_{min} & evict\ data\ packet \end{cases}$$

Additionally, our approach requires a generation timestamp field in every interest to measure ½ RTT. Furthermore, every data packet needs a field for the individually computed worst-case minimum caching period, e.g. an integer that defines the period in milliseconds. However, this is quite some overhead to both interest and data packet. If caching periods turn out to be similar, a default caching period may be defined as a global setting for every node. Only

in strongly deviating cases, an individual caching period shall be set in data packets to supersede the default setting. Moreover, CCN data packets no longer carry a nonce value. Therefore, it is required to re-introduce a nonce value field in CCN data packets. Moreover, `[eph,aggr]` data packets need an additional timestamp field. Table 5.9 summarizes the proceeding concerning permanent and short-term caching of results.

Table 5.9 – Permanent and Short-Term Caching of RO Results

[eph]	→	Short-term.
[eph,aggr]	→	Short-term, depending on further criteria.
[com]	→	Short-term.
[seq]	→	Permanent.
[prb]	→	Short-term.

Garbage Collection

With this information, garbage collectors can improve their proceeding. If memory gets short due to a large content store, a garbage collector can start evicting packets labeled with `[eph]`, `[com]`, and `[prb]`. These are simultaneously the three packet classes that can be purged periodically. If the situation is still strained, `[eph,aggr]` packets should be dropped next as they will be outdated sooner or later. Likewise, `[seq]` packets can eventually be dropped although they never lose their validity. However, once they are applied, further requests are no longer needed and therefore no longer expectable. `[syn]` and `[sem,X]` packets should be evicted only as a last resort. Without further information about time complexity of results, `[syn]` packets should be dropped before `[sem,X]` as the latter potentially involved effortful name reductions.

5.4.3 Mutable States

In conjunction with referentially opaque expressions, it needs to be clarified what mutable states are and how they work: A mutable state is bound to a specific entity, e.g. a node, a replica or a client. The state itself does not need to have a public name because it cannot be accessed directly from outside. The only way to interact with the state is controlled over routines that are associated with the state. Routines are bound to the state, i.e. pinned. Hence, they cannot travel around like other routines. These routines, however, need a (globally) routable name that is unique within an environment.

A mutable state exposes at least a *read* function that can be `[eph]` or `[eph,aggr]`. Read functions of mutable states never create copies without an additional discriminative element such as a nonce or timestamp. If a state shall be writable from outside the entity, an *update* function may be exposed as well. The update function can be `[com]`, `[seq]`, or `[prb]`. Depending on the synchronization mechanism, the mutable state may expose further routines, e.g. to *acquire* the resource for a sequence of updates before it is *released* again. Note that `[com]` and `[seq]` only need an internal mutex to avoid write collisions.

5.5 Extended Execution Model

Like subsection 2.2.2 that presented the generalized CCN/NDN execution model, this final section of the chapter takes up all newly introduced protocol changes and presents them condensed in a flow chart. Moreover, Figure 5.4 contains several markers, pointing to design choices that are discussed below.

- **Marker [A]**: This first equivalence class fork effects different expression matchings against the content store, either over the name only, the name and further matching criteria (option), or over name and nonce. `[seq]` expressions, although opaque, need only to match the name without the nonce.

- **Marker [B]**: This equivalence class fork is needed that `[eph]`, `[com]` and `[prb]` expressions omit PIT matching. Although opaque, `[eph,aggr]` and `[seq]` expressions can be matched against the PIT by name.

- **Marker [C]**: Expressions that will be forwarded are added to the PIT by their arrival interface and their unreduced as well as their potentially reduced name. The interest is forwarded in direction of the reduced name. The equivalence class fork is needed because `[eph]`, `[com]`, and `[prb]` expressions must additionally store the nonce.

- **Marker [D]**: The evaluation of locally available routines differs only for `[prb]` expressions that must first acquire shared resources it will alter.

- **Marker [E]**: This equivalence class fork is needed to differentiate between permanently and temporarily cacheable results.

- **Marker [F]**: Incoming data packets are matched against the PIT like interests against the CS (see marker [A]).

- **Marker [G]**: Ultimately, PIT matching is not a mandatory step for correct operation. However, it can be a beneficial shortcut and potentially increases efficiency. As discussed in subsection 5.3.1, PIT matching enables the avoidance of unnecessary computations, lower latencies and economizes transmissions. Moreover, a PIT match in this place does not only mean that an equivalent interest has already been forwarded, but also that the interest cannot be reduced any further by this node. Thus, the whole analysis and processing steps can be omitted and the classic CCN/NDN scheme for PIT matches can be followed, i.e. immediate interest aggregation and waiting for data (dashed green arrow). Note that unreduced names must be aggregated in order not to lose potential equivalence information.

- **Markers [H], [I], and [J]**: Whenever a node has the capability to reduce an expression with the help of a locally available rule set X (marker [H]) or a locally available routine (marker [I]), it should do it. This should be repeated in a cyclic manner (marker [J]) as long as there are rule sets to apply or routines to evaluate. Reduced expressions may now be satisfied by the CS or aggregate with an existing PIT entry.

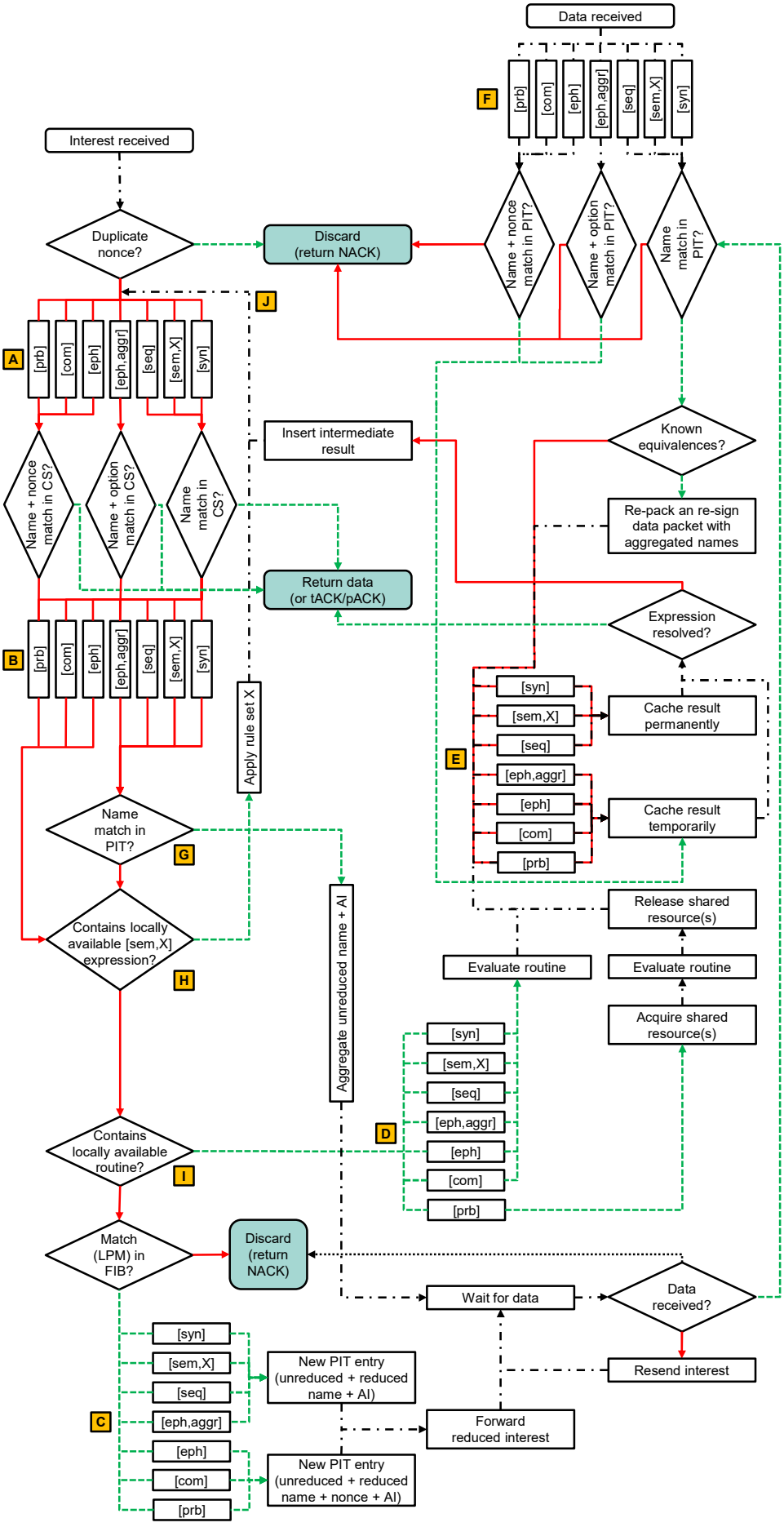
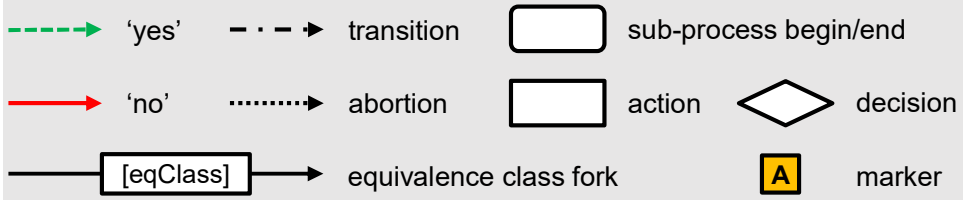

Data received
F
[prb]
[com]
[eph]
[eph,aggr]
[seq]
[sem,X]
[syn]
Interest received
Duplicate nonce?
Discard (return NACK)
Name + nonce match in PIT?
Name + option match in PIT?
Name match in PIT?
J
A
Insert intermediate result
Known equivalences?
Re-pack an re-sign data packet with aggregated names
Name + nonce match in CS?
Name + option match in CS?
Name match in CS?
Return data (or tACK/pACK)
Expression resolved?
B
Apply rule set X
E
Cache result permanently
Cache result temporarily
Name match in PIT?
G
Release shared resource(s)
Evaluate routine
Evaluate routine
Acquire shared resource(s)
Contains locally available [sem,X] expression?
H
Aggregate unreduced name + AI
D
Contains locally available routine?
I
Match (LPM) in FIB?
Discard (return NACK)
Wait for data
Data received?
Resend interest
C
New PIT entry (unreduced + reduced name + AI)
Forward reduced interest
New PIT entry (unreduced + reduced name + nonce + AI)

Figure 5.4 – Extended Execution Model

Boxes with round corners indicate beginning or ending of sub-processes while rectangles are actions. Rhombs indicate decisions. Arrows are flow lines. A dashed (green) arrow coming from a rhomb indicates 'yes' while a solid (red) arrow is 'no'. Morse code (black) arrows are transitions. Dotted (black) flow lines indicate process abortions. Additionally, rectangles with ECS tags that do not interpose but overlay arrows, are used to fork the flow according to equivalence classes.

Legend:



FORTRAN – "the infantile disorder" –, by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use.

PL/I --"the fatal disease"-- belongs more to the problem set than to the solution set.

- Edsger Dijkstra, 1975 (85)

6. Evaluation

The evaluation of the proposed approach is carried out by demonstrating several example applications that profit from different equivalence classes, focused on referentially opaque functions. For example, equivalence classes can be used to realize a simple programming language that integrates network level requests for contents and results in its code. Such programs are not bound to a single computer but can be evaluated distributed over an information-centric network (section 6.1). Remaining scenarios include outsourced counter replication (section 6.2), sensor data aggregation & side effects (section 6.3), as well as software defined networking (section 6.4).

Chapter	Section
Background	6.1 A Simple Mini Language
The Soft Spots	6.2 CmRDT: Outsourced Counter Replication
Equivalence Classes	6.3 An Edgy Smart Home: Sensor Data Aggregation & Side Effects
Protocol & Architecture Adaptions	6.4 SDN? Done!
Evaluation	
Related Work	

TL;DR – Key Messages

- ✓ The simple mini language integrates concepts of equivalence classes and differentiates between local and remote identifiers. When using a local identifier, the referenced object or routine declaration must be locally available, while when using remote identifiers, the object or result may also be fetched from remote during run-time.
- ✓ With the help of block expressions, i.e. sequentially composed expressions, it is possible to declare statement lambdas (a.k.a. multi-line / multi-statement lambdas). In contrast to expression lambdas (a.k.a. single-line / single-statement lambdas), statement lambdas can be used to declare comprehensible programs.

✓ To determine the overall equivalence class of block expressions, the same superseding rules as for nested expressions apply, i.e. the "worst" individual equivalence class is relevant.

6.1 A Simple Mini Language

Block Expressions

The main idea of the simple mini language (SML) is to mix imperative and functional programming concepts. In section 5.1, nesting has been discussed as a form of composed expressions. However, different from nesting, expressions can also be composed sequentially. Individual expressions that are arranged sequentially are often referred to as *statements*. We call a block of sequentially arranged statements/expressions a *block expression*. Such block expressions are the main building blocks of SML and introduce the imperative notion. The imperative syntax makes the language very easy to write and understand. Moreover, it offers a well-known approach to work with side effects, i.e. assignments. However, sequential composition does not immediately imply sequential evaluation. The attribution of equivalence classes reveals the necessary information where and when caching and concurrency is possible. Again, the question arises how equivalence classes of individual statements/expressions influence the overall equivalence class of a block expression. As it was the case for nesting, the "worst" individual equivalence class is relevant. Hence, Figure 5.1 and Table 5.1 also apply to block expressions.

Code Listing 6.1 shows an arbitrary extract from a block expression. The color coding is the same as in Figure 5.1. It contains several calls to expressions from different equivalence classes. Despite their sequential declaration, all parallelizable function calls, highlighted in green, can be evaluated concurrently without any need to wait for them to return. Whenever a problematic expression is next to evaluate, highlighted in red, all necessary shared resources must be acquired before the expression can be evaluated.

Code Listing 6.1 – Sequential Expression Composition / Block Expressions

This code listing shows an arbitrary sequential expression composition as it can appear in a block expression. Calls in red lines 3 and 6 must be evaluated sequentially. Calls in green lines can be parallelized, e.g. lines 4-5 and line 7. Dark green lines 4 and 7 can potentially profit from cached results. The overall block expression must be considered problematic because it contains problematic expressions.

```
1 <Start block expression>
2 …
3 /A/Problematic/Call()[prb];
4 /A/Transparent/Call(argExpr1[syn])[sem,X];
5 /An/Ephemeral/Call(argExpr1[syn])[eph];
6 /Another/Problematic/Call()[prb];
7 /A/Sequence-Idempotent/Call(argExpr1[syn])[seq];
8 …
9 <End block expression>
```

In contrast to *expression lambdas* (a.k.a. single-line / single-statement lambdas), block expressions help to declare *statement lambdas* (a.k.a. multi-line / multi-

statement lambdas). Such statement lambdas can be used to declare sequentially composed routines that can be shifted en bloc.

Routine Declarations

Declarations of new routines start with the 'decl' keyword, followed by the name/identifier of the routine. Next in row is the becomes sign '=', followed by a comma-separated list of identifiers between parentheses. These identifiers are parameter names for arguments of the routine. Parameters can be used in routine declarations without any further declaration/initialization. The parameter list is followed by the lambda operator '=>', which itself is followed by a block expression between braces. The block expression is the body of the routine and contains sequentially composed statements. Statements are delimited by the semicolon ';'. If the routine has a return value, the last statement must begin with the 'return' keyword. Routine declarations are finalized by a comma-separated list of attributes between brackets. At least the overall equivalence class of the composed expression must be indicated. The generic structure is given below in Code Listing 6.2.

Code Listing 6.2 – Generic Routine Declaration

The given routine is a function because it has a return value. The number of arguments, statements, and attributes is chosen arbitrarily. Apart from a return statement at the end of the body, action declarations are constructed identically.

```
1 DECL name = (pName1,pName2,pName3) =>
2 {
3    statement1;
4    statement2;
5    RETURN statement;
6 }[eqClass,attr2]
```

Type System

We decided to design SML as a *dynamically typed* language because this is a very natural fit. In dynamically typed languages, the name of a variable is bound to only one object, no matter what type the object has (86). In CCN/NDN, there are only data packet objects. Their payload, however, can be of a certain data type. The binding of the name to the object (~the referent) happens at run-time. Hence, the name can be bound to different objects at run-time, again no matter what type the payload has. This is essential when working with opaque functions that return different values on changed conditions.

Operators

The language supports classic binary, relation, unary, and Boolean operators. They are listed in Table 6.1. Note that the colon sign ':' is used as division operator because the forward slash '/' will be used differently.

Local and Remote Identifiers

One specialty of SML is the differentiation between local and remote identifiers for variables and routines. While local identifiers are normal alphanumeric constructs, remote identifiers are constructed like classic CCN/NDN content names. They consist out of several name components, delimited by the forward slash '/'. The idea is the following: When using a local identifier, the referenced object or routine declaration must be locally available, while when using remote identifiers, the object or result may also be fetched from remote during run-time. Remote means outside of the program memory. Hence, the remote retrieval location can be the content store of the executing node as well as any other node external resource.

Table 6.1 – Operators in SML

'cand' stands for 'conditional and'. It means that if the expression left from the operator is 'false', the overall expression immediately evaluates to 'false', without evaluating the expression on the right. The expression on the right is only evaluated if the expression on the left is 'true'. 'cor' means 'conditional or'. The overall expression immediately evaluates to 'true' if the expression on the left is 'true'. The expression on the right is only evaluated if the expression on the left is 'false'.

binary		relation		unary		Boolean	
+	plus	<	lt	!	not	&&	and
–	minus	>	gt	+	plus	||	or
*	times	<=	le	–	minus	&?	cand
:	division	>=	ge			|?	cor
%	modulo	==	eq				
		!=	ne				

Variables

Variables must be initialized explicitly. Hence, it is not possible to use a variable, i.e. to assign a value or to read from it, before it was declared. The declaration of variables starts with the 'var' keyword and is followed by an identifier. Local variables are easy to manage as they are allocated in the program memory of the executing node. In contrast, remotely accessible variables and associated functions to interact with it (see subsection 5.4.3) are allocated in the content store. An additional mechanism must ensure that no homonymous variable already exists in the environment.

Commands

For the statement body, common language constructs like assignment and call commands, as well as conditionals and while loops are available. Their syntax is listed in Table 6.2.

Table 6.2 – Commands in SML

An identifier can be a local or remote one. 'cond' stands for condition. These must be expressions that evaluate to a Boolean value.

Syntax	Command
identifier = expression	assignment
IF (cond) {…} ELSE IF (cond) {…} ELSE {…}	conditional
WHILE (cond) {…}	while loop
CALL identifier (arguments)	(remote) routine call

Writing to a remote identifier/variable is allowed. It can be done by calling the generic update function or by using the assignment command. However, the assignment command is only syntactic sugar for calling the update function.

```
/Name/Of/Var = value;
   ↓
CALL /Name/Of/Var/Update(value)[prb];
```

Syntax Examples

The section is closed with two small examples without any special abilities. Their sole intent is to demonstrate the syntax of SML. More sophisticated examples follow in the remaining sections of this chapter.

Code Listing 6.3 – SML Example 1: Factorial

Referentially transparent function with only local variables and a while loop. No inherent concurrency but cacheable and reusable results. The code can be shifted and executed wherever it suits best. The function itself can be called concurrently. There is no need to wait on the termination of another instance.

```
1  decl /ch/unibas/math_lib/Factorial = (x) =>
2  {
3     var fact;
4     fact = x;
5     while(x>1){x=x-1;fact=fact*x;};
6     return fact;
7  }[syn]
```

Code Listing 6.4 – SML Example 2: Heaviside Function

The Heaviside (step) function can be implemented with a simple conditional. The code is also mobile, and results can be cached and reused permanently.

```
1  decl /ch/unibas/math_lib/Heaviside = (x) =>
2  {
3     var result;
4     if(x<0)
5     {
6        result = 0;
7     }
8     else if(x>=0)
9     {
10       result = 1;
11    };
12    return result;
13 }[syn]
```

6.2 CmRDT: Outsourced Counter Replication

This example examines a scenario where a website (content + advertisement) is created dynamically on every new request. Moreover, each request for the website should be counted with a replicated counter. The example contains `[syn]`, `[eph]`, `[eph,aggr]` and `[com]` expressions. The scenario includes the following players and items:

- **Author 'Bob':** Bob wrote an interesting article that he wants to share. Additionally, he has two whishes: 1. He would like to know how many people requested his article. 2. He would like to earn some money with his article. Therefore, Bob decided to make the article accessible only over a function $f_{6.1}$ that 1. involves a replicated counter for that the article cannot be requested without being counted and 2. dynamically combines the article with a current advertisement. The corresponding SML code is given in Code Listing 6.5.

  $f_{6.1}$: `/An/Interesting/Article()[com]`

- **An interesting article:** This is some interesting text that has been written once. The text is finalized, hence, does not change over time and can therefore be considered immutable.

- **Intermediate node $N_{6.1}$:** possesses function $f_{6.1}$ that has been published by Bob, for example due to a prior request.

- **Web service provider, node $N_{6.2}$:** provides access to a function $f_{6.2}$ that composes two arguments, e.g. an article and an advertisement, to a HTML file that is returned.

  ```
  f6.2: /WebsiteServices/ComposeWebsite
          (argExpr1,argExpr2)[syn]
  ```

- **Advertisement service provider, node $N_{6.3}$:** delivers current advertisement banners over function $f_{6.3}$. Imagine, that this function is bound to the node, i.e. its code is immobile. The function can only be called. Tschudin and Sifalakis call such functions *pinned functions* (3).

  ```
  f6.3: /AdServices/GetCurrentAd()[eph]
  ```

- **Replicated counter:** Bob uses a replicated counter that is implemented as operation-based CmRDT. 3 replicas '$R_{6.1}$', '$R_{6.2}$', and '$R_{6.3}$' exist, all of them maintaining one individual counter c_1, c_2, and c_3. Moreover, all replicas expose a commutative increment function that is also pinned.

  ```
  f6.4: /CmRDT/Replica/1/Increment(argExpr1)[com]
  f6.5: /CmRDT/Replica/2/Increment(argExpr1)[com]
  f6.6: /CmRDT/Replica/3/Increment(argExpr1)[com]
  ```

  Furthermore, they feature pinned functions to read the current counter state.

  ```
  f6.7: /CmRDT/Replica/1/ReadCounter()[eph,aggr]
  f6.8: /CmRDT/Replica/2/ReadCounter()[eph,aggr]
  f6.9: /CmRDT/Replica/3/ReadCounter()[eph,aggr]
  ```

  The mutual update logic in order that all three counters will eventually converge is integrated in $f_{6.1}$ and discussed at the end of this section (see 'Note 3').

- **Website requester 'Alice':** Alice is interested in reading Bob's article. She only cares about the article. However, there is no other possibility for her than calling $f_{6.1}$ that includes advertisements and entails the counter increment. The advertisement is tangible for her, the counter increment is not.

Code Listing 6.5 – Outsourced Counter Replication

This code listing shows the declaration of $f_{6.1}$ that Bob created in order to publish his article.

```
 1 decl /An/Interesting/Article = () =>
 2 {
 3    var anInterestingArticle;
 4    anInterestingArticle = "This is interesting!";
 5    call /CmRDT/Replica/1/Increment(1)[com];
 6    call /CmRDT/Replica/2/Increment(1)[com];
 7    call /CmRDT/Replica/3/Increment(1)[com];
 8    return [/WebsiteServices/ComposeWebsite(
 9       /AdServices/GetCurrentAd()[eph],
10       anInterestingArticle
11    )[syn]][eph];
12 }[com]
```

Figure 6.1 depicts the distributed evaluation scenario that is explained next. The figure is accompanied by Table 6.3 that lists numbered requests and responses appearing in the scenario.

1. Alice wants to request the interesting article. To do so, she sends out interest $i_{6.1}$ for $f_{6.1}$, i.e. the function that Bob published to access his article.
2. $i_{6.1}$ is forwarded in direction of Bob. However, node $N_{6.1}$ already possesses $f_{6.1}$ from a prior request.
3. Node $N_{6.1}$ is willing to evaluate $f_{6.1}$.
4. According to $f_{6.1}$'s declaration (see Code Listing 6.5), three commutative function calls to increment three counters (lines 5-7, generating interests $i_{6.2}$, $i_{6.3}$, and $i_{6.4}$) as well as the wrapped ephemeral function call to compose the website (lines 8-11, generating interest $i_{6.5}$) can be requested *concurrently* from node $N_{6.1}$.
5. Interests $i_{6.2}$, $i_{6.3}$, and $i_{6.4}$ are forwarded in direction of replicas $R_{6.1}$, $R_{6.2}$, and $R_{6.3}$ that possess the called increment functions $f_{6.4}$, $f_{6.5}$, and $f_{6.6}$ that write to the associated counter states c_1, c_2, and c_3. Interests are answered by transient acknowledgments (tACKs, data packets $d_{6.2}$, $d_{6.3}$, and $d_{6.4}$) to confirm that they have been applied.
6. Interest $i_{6.5}$ is forwarded in direction of $f_{6.2}$ that is available at node $N_{6.2}$. There are two things to note. 1. The variable is not retrievable from outside the program memory. Hence, node $N_{6.1}$ must replace the local variable in the call with the value of the variable `anInterestingArticle`. In this case, applying call-by-value does not cause problems as the content of the variable is not a secret that must be protected. 2. $f_{6.2}$ itself is a referentially transparent function. Nevertheless, as it is called with an ephemeral argument expression, the composed expression is ephemeral, too.
7. Before node $N_{6.2}$ can evaluate $f_{6.2}$, it needs to call function $f_{6.3}$ that delivers a current advertisement banner. To do so, node $N_{6.2}$ sends interest $i_{6.6}$ to node $N_{6.3}$. $i_{6.6}$ is `[eph]` because the banner can change over time.
8. $N_{6.3}$ answers the request with data packet $d_{6.6}$ that is returned to node $N_{6.2}$.
9. Node $N_{6.2}$ holds now all necessary parts to evaluate $f_{6.2}$, or more precisely the expression from $i_{6.5}$. $N_{6.2}$ sends back data packet $d_{6.5}$, the HTML website consisting of the advertisement banner and Bob's article, to node $N_{6.1}$.
10. As soon as node $N_{6.1}$ has received all four responses $d_{6.2}$, $d_{6.3}$, $d_{6.4}$, and $d_{6.5}$, it returns the final result $d_{6.1}$ to Alice.

Table 6.3 – Outsourced Counter Replication

Requests (~interests) and responses (~data packets) in this table accompany the example scenario depicted in Figure 6.1.

Request	Response
$i_{6.1}$: /An/Interesting/Article()[com] {n_0xb8ac280b}	$d_{6.1}$: Same payload as $d_{6.5}$ but with the name and nonce of $i_{6.1}$.
$i_{6.2}$: /CmRDT/Replica/1/Increment(1)[com] {n_0x17fa68f9}	$d_{6.2}$: Transient acknowledgment (tACK). Name and nonce of $i_{6.2}$.
$i_{6.3}$: /CmRDT/Replica/2/Increment(1)[com] {n_0x7e9d7915}	$d_{6.3}$: Transient acknowledgment (tACK). Name and nonce of $i_{6.3}$.
$i_{6.4}$: /CmRDT/Replica/3/Increment(1)[com] {n_0x82e6622d}	$d_{6.4}$: Transient acknowledgment (tACK). Name and nonce of $i_{6.4}$.
$i_{6.5}$: [/WebsiteServices/ComposeWebsite (/AdServices/GetCurrentAd()[eph], "This is interesting!")[syn]][eph] {n_0x75f065f7}	$d_{6.5}$: HTML file with the article's text and the advertisement banner. Name and nonce of $i_{6.5}$.
$i_{6.6}$: /AdServices/GetCurrentAd()[eph] {n_0xe8a82e43}	$d_{6.6}$: Advertisement banner. Name and nonce of $i_{6.6}$.
$i_{6.7}$: /CmRDT/Replica/2 /ReadCounter()[eph,aggr]	$d_{6.7}$: Current counter state. Name of $i_{6.7}$ and at least a timestamp.

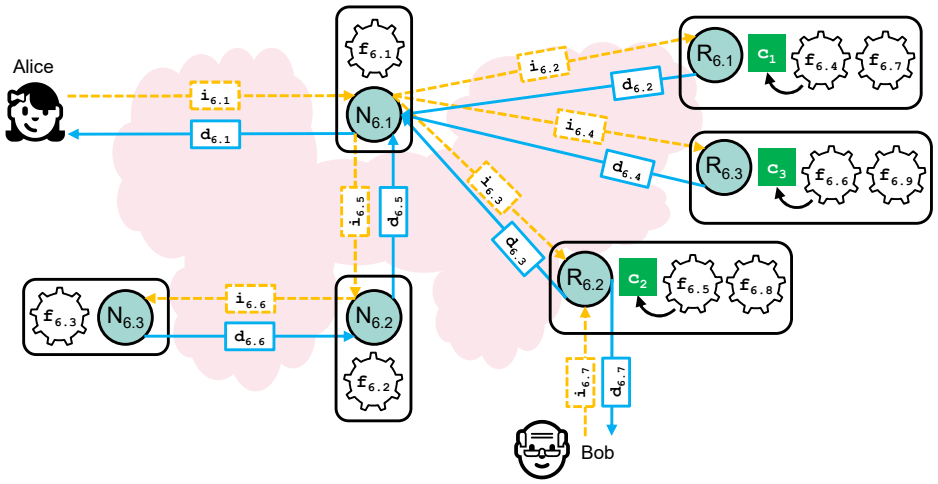
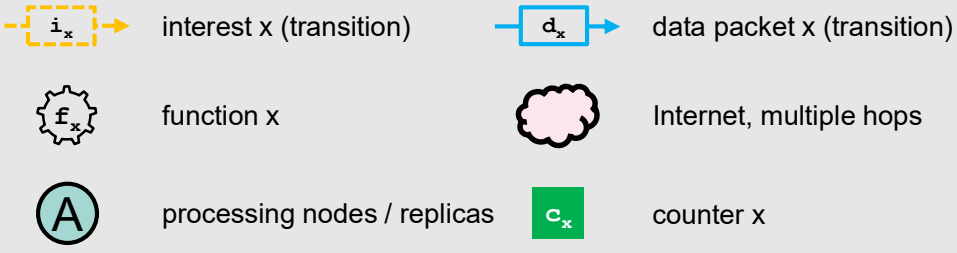


Figure 6.1 – Outsourced Counter Replication

The three individual counters c_1, c_2 and c_3 at $R_{6.1}$, $R_{6.2}$, and $R_{6.3}$ form a replicated counter by means of function $f_{6.1}$ that takes care of replica updates on each request. Bob is out the service provision cycle. He can sample the current counter state by requesting it from one of the replicas ($i_{6.7}$).

Legend:



Note 1: No data packet will be cached permanently because no expression is referentially transparent or sequence-idempotent. Every new call for $f_{6.1}$ has a new nonce and entails the same procedure as described above.

Note 2: Bob is out of the service provision cycle thanks to node $N_{6.1}$. From time to time, he can check how often his article has been requested by issuing interest $i_{6.7}$. Instead of replica $R_{6.2}$, Bob could have asked any other replica just as well. Obviously, CRDTs are only eventually consistent. Hence, response $d_{6.7}$ may not reflect an up-to-date / converged state. A second reason why requests to read a counter state can be aggregated (equivalence class `[eph,aggr]`) is that counters may not change dramatically during one round-trip time.

Note 3: Counters are in fact simple variables that expose an increment and a read function. They do not need to organize mutual information cycles about updates. The three individual counters are composed to a replicated counter by means of function $f_{6.1}$. In other words, the duty to distribute updates to all replicas has been outsourced to a function that is mobile and that can be executed from arbitrary nodes. It is therefore an example how to offload a task to the fog/edge.

6.3 An Edgy Smart Home: Sensor Data Aggregation & Side Effects

This example examines a scenario where sensor data is collected and aggregated. Depending on collected data, a device should be powered on or off. The example contains `[eph]`, `[eph,aggr]` and `[prb]` expressions. Apart from side effects, a loop and conditionals will be used. The scenario includes the following players and items:

- **Heating:** The central heating $H_{6.1}$ in Alice's house has two soft switches: the main switch and the heat switch. Both can be set to 'true' ("on") or 'false' ("off"). The heating will run only if both switches are 'true'. The main switch determines if the heating should run in general or not at all, independent of the temperature. The heat switch is "on" if the house temperature is below a certain target temperature and "off" if it is above the target temperature. Main switch and heat switch both feature a problematic 'update' and an ephemeral 'read' function.

  $f_{6.11}$: `/My/Home/Heating/MainSwitch/Update(argExpr1)[prb]`
  $f_{6.12}$: `/My/Home/Heating/MainSwitch/Read()[eph]`

  $f_{6.13}$: `/My/Home/Heating/HeatSwitch/Update(argExpr1)[prb]`
  $f_{6.14}$: `/My/Home/Heating/HeatSwitch/Read()[eph]`

  The target temperature can also be updated and read.

  $f_{6.15}$: `/My/Home/Heating/TargetTemp/Update(argExpr1)[prb]`
  $f_{6.16}$: `/My/Home/Heating/TargetTemp/Read()[eph]`

- **Minicomputers:** A first minicomputer $M_{6.1}$ is next to the heating and permanently runs the main heating control function $f_{6.10}$ that contains the program logic. Its declaration is given in Code Listing 6.6.

  ```
  f6.10: /My/Home/Heating/Control()[prb]
  ```

  A second minicomputer $M_{6.2}$ closes the gap between heating and sensors, e.g. because they cannot reach each other directly due to massive concrete walls. $M_{6.2}$ possesses function $f_{6.17}$ to calculate the average current temperature in the house. $f_{6.17}$'s declaration can be found in Code Listing 6.7.

  ```
  f6.17: /My/Home/AvgTemp()[eph,aggr]
  ```

- **Sensors:** There are three sensors, one in the kitchen ($T_{6.1}$), one in the living room ($T_{6.2}$) and another one in the office ($T_{6.3}$). All of them expose a 'read' function that is `[eph,aggr]` because room temperatures do not change significantly within a short time.

  ```
  f6.18: /My/Home/Kitchen/TempSensor/Read()[eph,aggr]
  f6.19: /My/Home/LivingRoom/TempSensor/Read()[eph,aggr]
  f6.20: /My/Home/Office/TempSensor/Read()[eph,aggr]
  ```

- **Alice and her mobile:** Alice installed a smart home app on her mobile. The app enables her to 1. turn the main switch of the heating "on" and "off" as well as to read the switch state, 2. update and read the target temperature, and 3. to read the average current temperature in her house. Hence, the app enables her to call functions $f_{6.11}$ and $f_{6.12}$, $f_{6.15}$ and $f_{6.16}$, and $f_{6.17}$.

Code Listing 6.6 – An Edgy Smart Home: Heating Control Function $f_{6.10}$

This code listing shows the declaration of the heating control function $f_{6.10}$ which is running on minicomputer $M_{6.1}$. Assume that SleepMinutes(x) is a local library function that keeps busy the current thread for x minutes. Furthermore, note that updating switch states (lines 8 and 10) are problematic operations because they have side effects. They are idempotent operations, however, they are not sequence-idempotent.

```
 1 decl /My/Home/Heating/Control = () => {
 2   var avgTemp;
 3   while(true){
 4     if(/My/Home/Heating/MainSwitch/Read()[eph]){
 5       avgTemp = /My/Home/AvgTemp()[eph,aggr];
 6       targetTemp = /My/Home/TargetTemp/Read()[eph];
 7       if(avgTemp > targetTemp){
 8         call /My/Home/Heating/HeatSwitch/Update("off")[prb];}
 9       else if(avgTemp <= targetTemp){
10         call /My/Home/Heating/HeatSwitch/Update("on")[prb];};
11       call SleepMinutes(5);}
12     else{
13       call SleepMinutes(5);};
14   };
15 }[prb]
```

Code Listing 6.7 – An Edgy Smart Home: Sensor Data Aggregation Function $f_{6.17}$

This code listing shows the declaration of $f_{6.17}$ that is located on minicomputer $M_{6.2}$. It retrieves measured temperatures concurrently from sensors $T_{6.1}$ in the kitchen, $T_{6.2}$ in the living room, and $T_{6.3}$ in the office and aggregates the results to an average temperature. The return expression cannot be evaluated until all three variables 'temp1', 'temp2', and 'temp3' can be read. These three *shared resources* have been acquired by assignment commands in lines 6-8. The shared resources are not released before the values have been assigned, i.e. written. The total function is `[eph,aggr]` although it contains assignment commands. However, they are all local. Hence, the function does not change any state outside of its scope.

```
1  decl /My/Home/AvgTemp = () =>
2  {
3     var temp1;
4     var temp2;
5     var temp3;
6     temp1 = /My/Home/Kitchen/TempSensor/Read()[eph,aggr];
7     temp2 = /My/Home/LivingRoom/TempSensor/Read()[eph,aggr];
8     temp3 = /My/Home/Office/TempSensor/Read()[eph,aggr];
9     return (temp1+temp2+temp3):3;
10 }[eph,aggr]
```

Figure 6.2 depicts the distributed smart home scenario whose steps are explained next. The figure is accompanied by Table 6.4 that lists numbered requests and responses appearing in the scenario.

1. Initially, the heating's main switch is set to 'false'. Alice wants to check the average current temperature over her smart home app. To do so, she sends out request $i_{6.8}$ which is forwarded to minicomputer $M_{6.2}$ where the corresponding function $f_{6.17}$ resides.
2. Function $f_{6.17}$ on minicomputer $M_{6.2}$ requests temperatures from the sensors in the kitchen $T_{6.1}$, living room $T_{6.2}$, and office $T_{6.3}$ concurrently by sending out interests $i_{6.9}$, $i_{6.10}$, and $i_{6.11}$.
3. The sensors evaluate functions $f_{6.18}$, $f_{6.19}$, and $f_{6.20}$ that read the current sensor values (18°C, 20°C, 19°C) and send back the information to $M_{6.2}$ as data packets $d_{6.9}$, $d_{6.10}$, and $d_{6.11}$.
4. $f_{6.17}$ can now evaluate the return statement. The result, 19°C, is returned to Alice's mobile as data packet $d_{6.8}$.
5. For Alice, this is a little bit too cold. Thus, she sets the main switch of the heating $H_{6.1}$ to 'true' by issuing interest $i_{6.12}$. $i_{6.12}$ triggers function $f_{6.11}$ on the heating that updates the main switch state to 'true'. Her request is answered with a transient acknowledgment $d_{6.12}$.
6. Likewise, Alice sets the target temperature to 21°C by sending $i_{6.13}$ to the heating $H_{6.1}$. The request triggers $f_{6.15}$ that updates the target temperature state to the desired value. $i_{6.13}$ is also answered with a transient acknowledgment $d_{6.13}$.
7. Minicomputer $M_{6.1}$ runs the heating control function $f_{6.10}$ that probes every five minutes if it must take actions. Imagine that $i_{6.14}$ is the first probing interest after Alice turned "on" the main switch. Accordingly, the heating will answer the request with the result 'true' ($d_{6.14}$).

8. As the condition resolved to 'true', $f_{6.17}$ enters the if-block (Code Listing 6.6, lines 4/5). According to the code, $M_{6.1}$ concurrently requests average current temperature from $M_{6.2}$ ($i_{6.15}$) and target temperature of the heating ($i_{6.19}$). Like Alice's initial request $i_{6.8}$, $i_{6.15}$ entails concurrent sensor readings ($i_{6.16}$, $i_{6.17}$, $i_{6.18}$) whose results ($d_{6.16}$, $d_{6.17}$, $d_{6.18}$) are aggregated to the average current temperature (still 19°C) that is returned from $M_{6.2}$ to $M_{6.1}$ ($d_{6.15}$). Meanwhile, the heating should have answered interest $i_{6.19}$ with data packet $d_{6.19}$ that contains the target temperature 21°C.
9. As soon as $M_{6.1}$ received average and target temperature, it compares them to each other. Because the average temperature is below the target temperature, the first condition (Code Listing 6.6, line 7) resolves to 'false'. However, the second condition (line 9) is checked, too, and resolves to 'true'. Accordingly, the last interest $i_{6.20}$ is sent from minicomputer $M_{6.1}$ to the heating $H_{6.1}$, triggering function $f_{6.13}$ that updates the heat switch to 'true'. The heating acknowledges the request with a tACK ($d_{6.20}$). As both switches are now 'true', the heating starts to heat the house.

Table 6.4 – An Edgy Smart Home

Requests (~interests) and responses (~data packets) in this table accompany the example scenario depicted in Figure 6.2.

Request	Response
$i_{6.8}$: /My/Home/AvgTemp()[eph,aggr]	$d_{6.8}$: The current average temperature, i.e. 19°C. Name of $i_{6.8}$ and at least a timestamp.
$i_{6.9}$: /My/Home/Kitchen/TempSensor /Read()[eph,aggr]	$d_{6.9}$: The temperature in the kitchen, i.e. 18°C. Name of $i_{6.9}$ and at least a timestamp.
$i_{6.10}$: /My/Home/LivingRoom/TempSensor /Read()[eph,aggr]	$d_{6.10}$: The temperature in the living room, i.e. 20°C. Name of $i_{6.10}$ and at least a timestamp.
$i_{6.11}$: /My/Home/Office/TempSensor /Read()[eph,aggr]	$d_{6.11}$: The temperature in the office, i.e. 19°C. Name of $i_{6.11}$ and at least a timestamp.
$i_{6.12}$: /My/Home/Heating/MainSwitch /Update(true)[prb] {n_0x3858624c}	$d_{6.12}$: A transient acknowledgment (tACK). Name and nonce of $i_{6.12}$.
$i_{6.13}$: /My/Home/Heating/TargetTemp /Update(21)[prb]{n_0xd90ec09c}	$d_{6.13}$: A transient acknowledgment (tACK). Name and nonce of $i_{6.13}$.
$i_{6.14}$: /My/Home/Heating/MainSwitch /Read()[eph]{n_0xd9e1951b}	$d_{6.14}$: The current main switch state, i.e. 'true'. Name and nonce of $i_{6.14}$.
$i_{6.15}$: /My/Home/AvgTemp()[eph,aggr]	$d_{6.15}$: The current average temperature, i.e. 19°C. Name of $i_{6.15}$ and at least a timestamp.
$i_{6.16}$: /My/Home/Kitchen/TempSensor /Read()[eph,aggr]	$d_{6.16}$: The temperature in the kitchen, i.e. 18°C. Name of $i_{6.16}$ and at least a timestamp.
$i_{6.17}$: /My/Home/LivingRoom/TempSensor /Read()[eph,aggr]	$d_{6.17}$: The temperature in the living room, i.e. 20°C. Name of $i_{6.17}$ and at least a timestamp.
$i_{6.18}$: /My/Home/Office/TempSensor /Read()[eph,aggr]	$d_{6.18}$: The temperature in the office, i.e. 19°C. Name of $i_{6.18}$ and at least a timestamp.
$i_{6.19}$: /My/Home/Heating/TargetTemp /Read()[eph]{n_0xf060fb20}	$d_{6.19}$: The target temperature, i.e. 21°C. Name and nonce of $i_{6.19}$.
$i_{6.20}$: /My/Home/Heating/HeatSwitch /Update(true)[prb] {n_0xad40d11e}	$d_{6.20}$: A transient acknowledgment (tACK). Name and nonce of $i_{6.20}$.

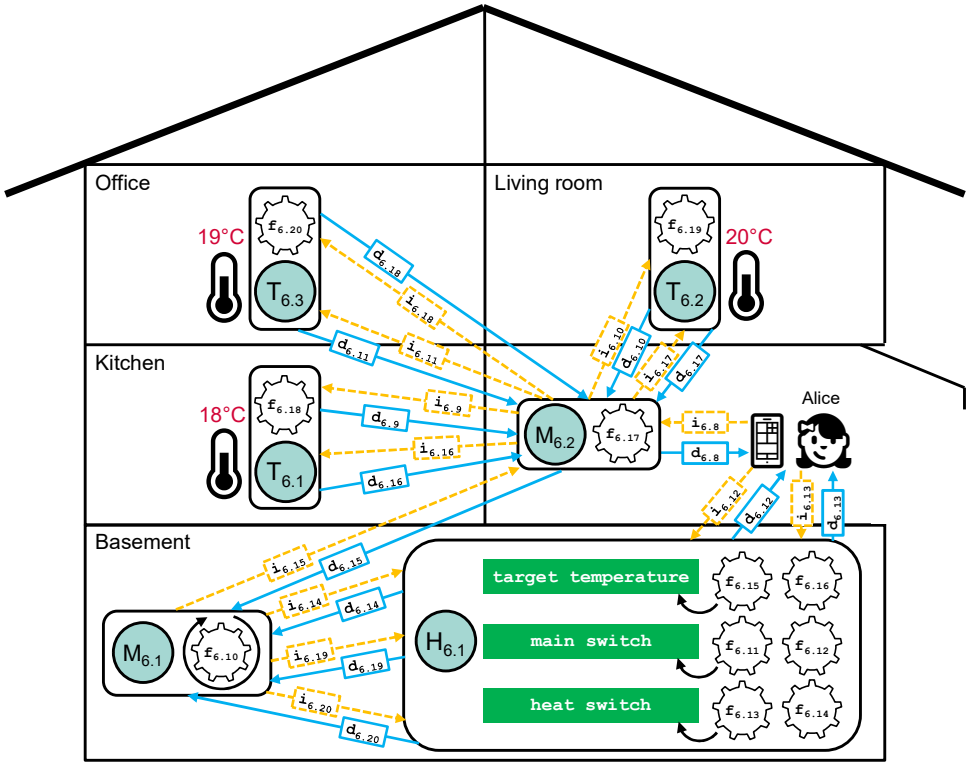
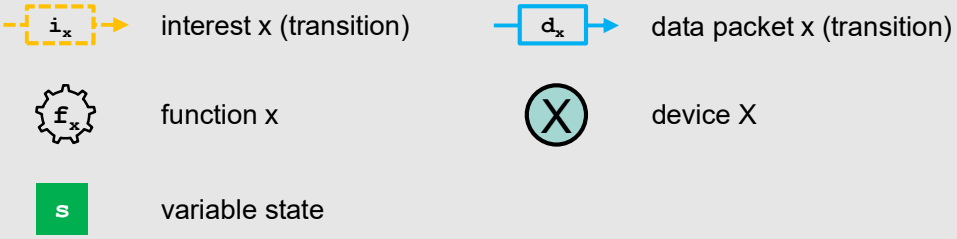


Figure 6.2 – An Edgy Smart Home

Alice's house contains the following devices: $H_{6.1}$ – central heating, $T_{6.1}$ / $T_{6.2}$ / $T_{6.3}$ – temperature sensors, $M_{6.1}$ / $M_{6.2}$ – minicomputers, and Alice's mobile. Callable functions on these devices are: $f_{6.10}$ – heating control, $f_{6.11}$ – updating the main switch, $f_{6.12}$ – reading the main switch, $f_{6.13}$ – updating the heat switch, $f_{6.14}$ – reading the heat switch, $f_{6.15}$ – updating the target temperature, $f_{6.16}$ – reading the target temperature, $f_{6.17}$ – sensor data aggregation, $f_{6.18}$ – reading sensor $T_{6.1}$, $f_{6.19}$ – reading sensor $T_{6.2}$, $f_{6.20}$ – reading sensor $T_{6.3}$.

Legend:



Note: No data leaves the house, not even over Alice's mobile. No (commercial) cloud service providers are involved in this edge-only scenario. Hence, a high level of privacy is already guaranteed. However, transmissions should of course be secured as well as the access to the heating should be controlled. For example, no one else than Alice should be able to change the target temperature.

6.4 SDN? Done!

SDN in a Nutshell

An already mentioned use case of equivalence classes is software defined networking (SDN) because it heavily depends on referentially opaque operations. SDN is generally understood as the separation of control plane and data plane. The data plane is responsible for forwarding packets from their source towards their destination according to installed rules. In contrast, the control plane defines and installs those forwarding rules. Hence, it is the control plane that decides where network traffic goes, the data plane just attends to orders. Besides routing decisions, the control plane can make decisions on firewall rules, load balancing (traffic to multiple destinations) and congestion control (traffic over multiple paths to one destination). As it would be too laborious to have per-packet rules or to request an action for each packet, rules are normally defined on a reduced set of packet properties. Often, rules are defined for a specific source/destination pair. Such rules are referred to as *flow rules* because a complete flow of packets between endpoints is concerned.

In SDN, these two planes are no longer on the same device. Switches house data planes while dedicated servers, i.e. controllers, take duties and responsibilities of the control plane. This has the advantage that cheap, efficient, and task-specific hardware can be used. Moreover, it allows to centrally control large networks. Data traffic gets programmable and can be guided dynamically through a network. Involved hardware can be configured remotely.

This example examines a scenario where an initial interest should travel from a source to a destination. Unfortunately, some network switches do not know where to forward the packet. Hence, they must ask the controller for forwarding information. In a later stage, Bob will reattach to the network in a different location what will necessitate to update forwarding information bases (FIBs) on some switches. The example contains `[syn]`, `[eph]` and `[prb]` expressions. The scenario includes the following players and items:

- **Switches $S_{6.21}$-$S_{6.30}$:** The switches form a network between Alice and Bob. Each of them contains a forwarding information base (FIB_{21}-FIB_{30}) that associates a name with an interface to a neighboring switch, i.e. the next hop in direction of the name. Moreover, each switch possesses a function that, when called, updates the FIB.

  f_X: `/Switch/X/FIB/Update(name,nextHop)[prb]`

  The function is problematic because it alters external mutable state, i.e. the FIB, in a non-commutative and non-sequence-idempotent way. The update function extends the FIB or overwrites existing entries. Thus, replies cannot be cached permanently.

- **Controller:** The SDN controller $O_{6.1}$ is the central contact point for the organization of the network. It contains a remotely accessible function $f_{6.31}$ that delivers up-to-date routing information on demand.

  $f_{6.31}$: `/Controller/Request/FwdRule(switchId,packet)[eph]`

The function produces ephemeral results. Forwarding rules depend on outer conditions, e.g. the network configuration or congestion, and are therefore quite volatile. Expressions calling function $f_{6.31}$ should therefore be marked with the `[eph]` ECS tag.

Figure 6.3 illustrates the network of switches and the attached controller. Table 6.5 that lists numbered requests and responses appearing in the scenario accompanies the figure. The scenario is the following:

1. Alice is attached to the network of switches over switch $S_{6.22}$. She tries to retrieve some immutable data from Bob by sending an `/Interest/Towards/Bob[syn]` ($i_{6.21}$).
2. $S_{6.22}$ has no forwarding information that matches the name of $i_{6.21}$. Instead of NACK'ing the interest, the switch contacts controller $O_{6.1}$ to request the needed forwarding information. To do so, $S_{6.22}$ calls function $f_{6.31}$ by issuing interest $i_{6.22}$. The function takes two arguments. Both are needed for that the controller can give adequate assistance. The first argument is the identifier of the switch and is needed because the forwarding information for an identical name differs from switch to switch. The second argument is the deliverable packet itself for that the controller can extract the necessary information. In our case, this is the content name.
3. The controller returns data packet $d_{6.22}$, containing the information that $S_{6.22}$ should forward $i_{6.21}$ over interface 2 towards $S_{6.25}$.
4. Like $S_{6.22}$, $S_{6.25}$ does not have a FIB entry that matches $i_{6.21}$. Hence, it also asks controller $O_{6.1}$ for help ($i_{6.23}$).
5. The controller returns data packet $d_{6.23}$, containing the information that $S_{6.25}$ should forward $i_{6.21}$ over interface 2 towards $S_{6.27}$.
6. From $S_{6.27}$, $i_{6.21}$ finds its way over $S_{6.29}$ to Bob who answers the interest with $d_{6.21}$.
7. In a later stage, Bob changes his network location. He is no longer attached to $S_{6.29}$ but to $S_{6.30}$. The controller, after having perceived this topological change, subsequently updates the FIBs of affected switches. This is done by issuing $i_{6.24}$, calling the FIB update function $f_{6.29}$ on switch $S_{6.29}$. An old FIB entry, saying that an `/Interest/Towards/Bob[syn]` should be forwarded over interface 1, is overwritten such that future interests for that content are henceforth forwarded over interface 2. The request is problematic and is therefore answered with a transient acknowledgment ($d_{6.24}$).
8. Exemplary, the same update proceeding is illustrated for switch $S_{6.30}$ whose update function $f_{6.30}$ is called with interest $i_{6.25}$. The interest is satisfied by data packet $d_{6.25}$. Other switches that need to be updated can be dealt with in the same way.

Note 1: A minimum requirement for this scenario is that all switches know at least how to reach the controller, i.e. a next hop closer to it. Otherwise, they would not be able to request information from the controller.

Note 2: Other SDN tasks like firewall rules, load balancing and multi-path routing can be implemented with problematic functions, similar to FIB update functions $f_{6.21}$-$f_{6.30}$.

Table 6.5 – SDN? Done!

Requests (~interests) and responses (~data packets) in this table accompany the example scenario depicted in Figure 6.3.

Request	Response
$i_{6.21}$: /Interest/Towards/Bob[syn]	$d_{6.21}$: Some immutable data.
$i_{6.22}$: /Controller/Request /FwdRule($S_{6.22}$,$i_{6.21}$)[eph] {n_0x2f3fe301}	$d_{6.22}$: A FIB entry to install, saying that $i_{6.21}$ should be forwarded over interface 2 (towards $S_{6.25}$). Name and nonce of $i_{6.22}$.
$i_{6.23}$: /Controller/Request /FwdRule($S_{6.25}$,$i_{6.21}$)[eph] {n_0xf276f288}	$d_{6.23}$: A FIB entry to install, saying that $i_{6.21}$ should be forwarded over interface 2 (towards $S_{6.27}$). Name and nonce of $i_{6.23}$.
$i_{6.24}$: /Switch/9/FIB/Update("/Interest/Towards/Bob[syn]",2)[prb] {n_0x8293501d}	$d_{6.24}$: A transient acknowledgment (tACK). Name and nonce of $i_{6.24}$.
$i_{6.25}$: /Switch/10/FIB/Update("/Interest/Towards/Bob[syn]",2)[prb] {n_0x75333d3d}	$d_{6.25}$: A transient acknowledgment (tACK). Name and nonce of $i_{6.25}$.

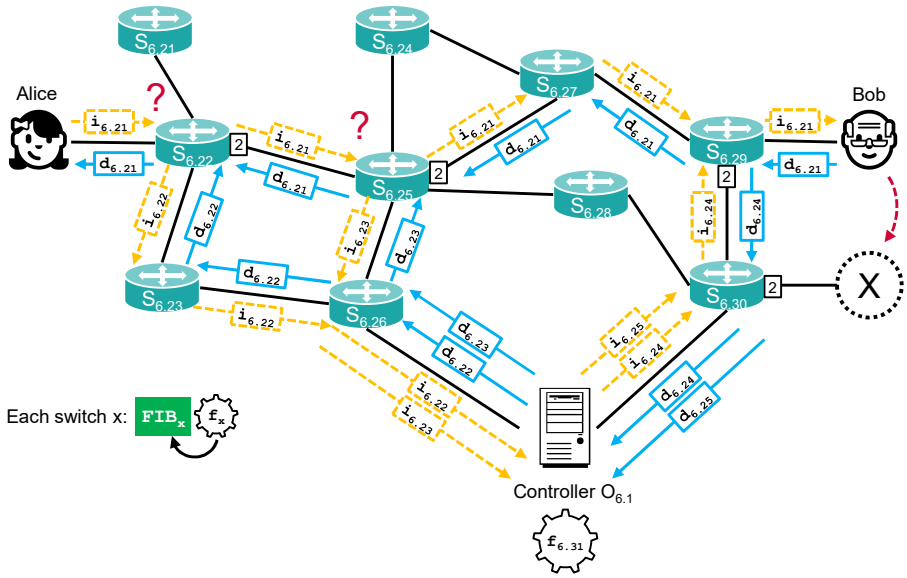
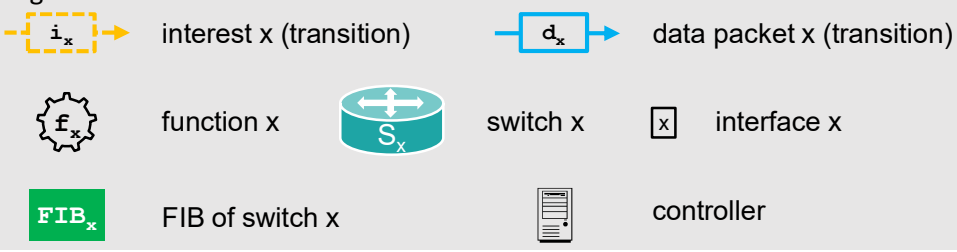


Figure 6.3 – SDN? Done!

The figure shows only involved interface numbers that are attributed according to the following rule: Interfaces of each switch are numbered clockwise from 1 to n, starting at 12 o'clock position.

Legend:



It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

- Edsger Dijkstra, 1975 (85)

7. Related Work

Although name-based computations gained some momentum through the advent of fog and edge computing, related work to NFN is still rare. Considerations about different types of expressions in this relation are even scarcer. Instead of treating many topics superficially, three closely related works that are doing name-based computations are inspected thoroughly. They are compared to our equivalence classes approach and mainly focus on the questions how they treat referential opacity (if at all), how they make use of names to reach their design goals, as well as how caching of results and their retrieval works. Because referential opacity is covered insufficiently in the selected works, an additional section with consideration on dynamic retrieval of mutable data helps to place our work in a broader landscape.

The chapter starts with looking at Remote Method Invocation in ICN (RICE) that tries to give a protocol-like, framework independent idea of how to do computations in CCN/NDN (section 7.1), followed by Named Function as a Service (NFaaS) that is an explicit implementation making use of unikernels (section 7.2). Service Centric Networking (SCN) is examined in section 7.3. The chapter ends with a brief section on HTTP caching and RESTful web services (section 7.4).

Chapter	Section
Background	7.1 RICE: Remote Method Invocation in ICN
The Soft Spots	7.2 NFaaS: Named Function as a Service
Equivalence Classes	7.3 SCN: Service Centric Networking
Protocol & Architecture Adaptions	7.4 HTTP Caching & RESTful Web Services
Evaluation	
Related Work	

TL;DR – Key Messages

- ✓ Remote Method Invocation in ICN (RICE) is the only approach that actively differentiates between referentially transparent and referentially opaque method invocations. Their approach to dodge CS and PIT matches (~aggregation), but not caching of results, is to use a random name component. We discussed the idea in subsection 5.4.1. Furthermore, RICE uses a 4-way handshake to initiate computations. During this phase, clients are authenticated, and parameters are passed. Requesting parameters instead of pushing them to providers resonates with CCN/NDN's attitude to never push data.
- ✓ Named Function as a Service (NFaaS) relies on unikernels, i.e. lightweight virtual machines, and is serverless. Hence, clients manage state that is involved in computations and send copies along with stateless requests. Function calls include an explicit `/exec` name prefix to distinguish them from requests to download function objects. Further prefixes like `/delay` or `/bandwidth` give additional hints for the execution machinery how interests should be treated. A score function decides whether kernels are called remotely or downloaded to be applied locally.
- ✓ Service Centric Networking (SCN) is an overlay on top of CCN/NDN and supports services. Everything is an object: content objects, service objects and combined content/service objects. Service objects cannot move by themselves. They can only be placed in appropriate locations. In order that a shared state is accessed only under controlled conditions, SCN proposes to establish sessions.
- ✓ The HTTP generally assumes transient contents and results. Its stateless methods are categorized into safe, idempotent and cacheable methods. This characterization is often used in RESTful web services when implemented over HTTP because RESTful web services must be clear about caching of responses. Cache control directives, generally present in responses, help to supply clients with valid results. Directives like 'no-cache', 'no-store', 'must-revalidate', and 'immutable' instruct intermediate caches how to handle cached contents and what to do when new requests occur, e.g. to mandatorily revalidate cached contents before sending a response back to a requester.

7.1 RICE: Remote Method Invocation in ICN

Referential Transparency and Opacity

A very recent work from Król et al. (87) presents an approach for remote method invocation in ICN (RICE). The approach should be universal, i.e. not being specifically tied to NFN or NFaaS (see section 7.2). Generally, the paper cares a lot about timings, especially about network timescale and application timescale. Moreover, the paper discusses client authentication and parameter passing issues. Nevertheless, the most interesting point concerning this thesis is that the authors distinguish referentially transparent from referentially opaque functions. In fact, they resort to the same trick as used by Shi (75) that was introduced in subsection 3.3.1. The requester is responsible to append a

random name component, making the name for every method invocation unique. Consequently, interests should not aggregate and not match any cached content. A discussion about the need to distinguish random name components from preceding name components as we provided in subsection 5.4.1 is missing, as well as a deeper inspection of referential opacity.

Król et al. (87) propose to individually retrieve large input parameters (we called them argument expressions) instead of passing them together with the interest. The way how they will be transferred to a producer is explained below. However, the interesting point is how parameters are reflected in the name of a referentially transparent request, namely by appending the hash over the input parameters to the name. Obviously, this hash is not random but equal for every call with syntactically equivalent input parameters. Hence, even without explicit input parameters, equal interests can aggregate and match the same cached content. The downside, however, is obvious, too. It frustrates to determine any other equivalence over argument expressions than the syntactic equivalence because even semantically equivalent expressions lead to different hashes.

Furthermore, RICE is not able to distinguish opaque from transparent results/data packets. All of them are cached likewise. Due to the random name component for opaque results, it is assumed that cached content will never be matched and therefore be dropped eventually. However, this is a waste of memory and should be omitted. Our equivalence class tagging is also present in data packets and can be used to purge transient results early. Finally, the authors perceive referential opacity only as different results for the same input parameters. There is no differentiation between return value and side effect, the latter not being mentioned at all.

4-Way Handshake

The provision of input parameters and client authentication are approached with a 4-way handshake. An interest from a consumer to a producer is not answered with data immediately. In between, there is an additional interest/data exchange that takes over some tasks. It is initialized by the producer and therefore referred to as *callback*. For example, it can be used to request (large) input parameters that has been excluded from the initial request. The authors argue that transmitting input arguments together with an interest contradicts CCN/NDN's flow balance principle, saying to never push data if not requested. Moreover, it facilitates content-based access control and encryption. The callback mechanism can also be used to negotiate a symmetric key that is subsequently used to establish a secure communication between consumer and producer.

Furthermore, the callback mechanism can be used to authenticate clients and to apply an according authorization policy. Especially, if a requested computation is labor-intensive, a producer does not want its function to be called by random clients. (D)DoS attacks would be very easy. Decoupling client authentication from the actual computation and the remaining communication between client and producer has the advantage that authentication must be done only once. Thereafter, efficient security tokens can be used. An alternative would be to sign interests. However, validating signatures and certificates for every packet in the communication may induce noticeable load on servers.

Network vs. Application Timescale

The authors present several approaches for dynamic content retrieval. The preferred approach is to use thunks. Król et al. (87) use them almost the same way as originally proposed by Sifalakis et al. (38). A thunk is immediately

created upon the reception of a request for a computation. While the thunk is sent back to the requester, the computation can already be started. In both cases, thunks contain a temporary name to retrieve the result and a completion time estimation. However, Król designed thunks as normal data packets that consume all matching PIT entries on its downstream path. A requester must request the result with the temporary thunk name after the completion time estimation elapsed. Thus, a remote method invocation consists of two separated phases: an initiation and the retrieval of the result. In contrast, Sifalakis designed thunks as an additional packet type that does not erase PIT entries but updates their timers. Król's approach has the advantage that intermediate nodes do not have to maintain any PIT state during the computation. This can be relevant for highly frequented intermediate nodes, especially in conjunction with long running computations whose PIT entries would remain for a long time. Contrariwise, identical interests cannot aggregate in the network. All requests advance to the server what may be a problem as well. Moreover, a ready result cannot be sent back to the requester without a request for the thunk name. Thus, 4 packets must be transmitted in minimum. The approach from Sifalakis has the advantage that interests can aggregate in the network, potentially unburdening servers. Moreover, a result can be returned without an additional request. Hence, 3 packets are enough in the best case. The price to pay is that intermediate nodes must manage more state on average because all ongoing computations remain in the PIT. Nevertheless, both approaches solve the problem that applications and involved computations require a different timescale (application timescale) than traditional requests for content that should not take much longer than an average RTT (network timescale).

7.2 NFaaS: Named Function as a Service

Named Function as a Service (NFaaS) from Król and Psaras (88) is more a concrete implementation that focuses on practical aspects than a theoretical approach to preform computations in CCN/NDN. Nevertheless, it necessarily contains theoretical considerations. Its main motivation is to serve new application demands such as minimum delays, e.g. for augmented reality, or to handle large amounts of data that flow in the reverse direction from traditional flows, e.g. generated by the IoT. Overall, this is a similar motivation as we provided in the background section. However, their solution approach is quite different from ours.

Special Name Components

NFaaS is an explicit extension of NDN to support in-network function execution. Therefore, it is fully interoperable with normal NDN nodes. Hence, not all nodes must be able to execute functions. NDN interests are adapted to request function execution and to include inputs. Function calls include an `/exec` name prefix to distinguish them from requests to download the function object. The `/exec` prefix can be followed by further extensions, giving additional hints for the execution machinery how the interest should be treated. The paper gives two examples. 1. A `/delay` name component saying that the computation is delay-sensitive and should therefore be executed as close to the edge as possible. 2. A `/bandwidth` name component indicating that the evaluation of the function is not delay-sensitive and therefore can take place a few hops from the edge. For example, this is a good strategy for large amounts of data that should not be transmitted the whole way to a cloud

computing center. Hence, name prefixes help to implement forwarding strategies to support certain scenarios, e.g. to decide whether to download a function or to forward the interest. Additionally, a user-specific hash over all input information is appended to the name of an interest, enabling to distinguish requests from different users. While the advantages of making interests user-specific are unclear, the disadvantages are obvious. Interest aggregation falls short completely and profiting from cached results is restricted to single clients that request the very same computation more than once. More interesting are a *task deadline* and a *discovery* field, also contained in interests. Discovery interests are flooded to the close neighborhood in order to find nodes that are willing to execute the requested computation. The task deadline is used to set PIT timers appropriately.

Serverless Unikernels

In NFaaS, functions are implemented as lightweight virtual machine (VMs) in form of explicitly named unikernels (89). Unikernels are bootable and immutable images, containing the application binary plus all required system components such as kernel and drivers (88). Unikernels are chosen to achieve system isolation, protecting the hosting OS and its filesystem. However, unikernels should be signed by their publishers. This enables executing nodes to verify signatures and to only execute trusted unikernels. Unikernels are stored in an additional data structure called *kernel store* (KS). Apart from storing function codes, the KS decides which functions to execute locally, based on a unikernel *score function*. The score function uses function popularity, i.e. request frequency, average hop counts of interests for that function, and other tuning parameters such as `/delay` and `/bandwidth` prefixes. The score helps to decide if it is worth to download the function or to forward the interest, with a tendency to download delay-sensitive functions, implicitly letting applications with higher bandwidth requirements diffuse towards the core of the edge domain.

Unikernels can be stored in any KS and shifted across the network without any restriction because they are independent of the execution environment. NFaaS is also designed such that there is no state information on the hosting node. In this so called *serverless* architecture, function state is managed by clients that are therefore called *rich clients*. Calls that depend or manipulate the same function state can be handled sequentially by different nodes without any handover process. A client appends the initial state to the interest. The updated state is sent back to the client in the resulting data packet and re-distributed from there. If state is large, it can be chunked and stored as named data. Instead of the state itself, the name of the data can be appended to an interest such that an evaluating node can request the required state. The same applies to large results. Only a name to retrieve the result is returned to the client. Before a client can again append the updated state to a subsequent interest, it must first download the updated state to locally apply the update, i.e. overwriting the previous function state.

This is in clear contrast to our approach where it is always the executing node that manipulates the state that is not necessarily stored on the same node as the client that called the function. A node that executes a side effecting routine must first acquire the critical section. It is either the data structure of the state itself that grants the access to the resource or a signaling mechanism inside the code. In NFaaS, there is no real shared resource such as a database that can be accessed by different clients. There is only function state and the

access to it is controlled by and limited to the function it is associated with. Accordingly, NFaaS does not differentiate between transparent and opaque routine calls. In NFaaS, it is always the state owning function that requests an update of a state, granting access to it at the same time, while in our approach, an arbitrary client can request an update. Additionally, NFaaS differs from our approach insofar that all data packets are cached, no matter if they are results of opaque and side effecting routine calls. However, the chance that another request hits a cached result is neglectable due to the user-specific hash name component. The positive side of the approach is that short-term caching is a non-issue. As all results are potentially cached permanently or at least long enough, chances that an expired and re-triggered interest causes a new computation are small.

7.3 SCN: Service Centric Networking

Service Centric Networking (SCN) from Braun et al. (90) proposes a generalization of CCN towards the support of services. It is positioned as an overlay on top of a CCN/NDN infrastructure. The paper talks specifically from CCNx (that became CICN (91)) as underlaying infrastructure because NDN was in its infancy at that time. However, there is no reason not to use NDN, too. Like RICE and NFaaS, SCN bases on an interest/data packet scheme. Hence, content or a service invocation is requested with an interest and the content or result is returned with a data packet. Through the support of services, it shall be possible to generate and manipulate contents. Hence, it is a holistic approach to service invocation and, for example, not limited to HTTP methods like Representational State Transfer (REST).

Functional Components and Processed Content

In this context, it is worth to mention that Braun et al. (90) points out an interesting feature of CCNx. Beyond simple retrieval of static data, it is possible to integrate *functional components* in interests to request *processed data*. The latest CCNx semantics and message TLV format documents from Mosko et al. (92), (93) call these components *application components*. They are not part of the content name but of the interest. They include a name that identifies an operation, i.e. a function, that is applied to the related named content. Optionally, the functional component can include function arguments. The sematic of functional components is application specific. Operations are always performed on the addressed content by the provider of the content. If the operation is unavailable in conjunction with the content, the request cannot be satisfied. Unfortunately, referential opacity and what it means to caching and interest aggregation remains undiscussed in all documents.

Object Orientation

Instead of only requesting static or processed content, SCN aims to be more flexible by allowing to address services (~functions) directly. The authors opine that everything should be an *object*, thus, there is no differentiation between content (object) and service (object). So-called *object names* should follow a *uniform naming scheme* for both services and contents. The paper, however, does not further specify how such object names look like. There are three object types. *Content objects* provide only one default read method that returns a certain content. The content is immutable because the object does not offer any further methods to write/manipulate the content. Hence, requesting SCN content objects resembles requesting immutable data packets in CCN/NDN. An interest with the name of a content object travels towards the object where

the read function is invoked. *Service objects* contain only functions that are not bound to any data. A user that invokes the service must specify the input data by either directly providing the contents or by indicating the location of the input data as additional parameters to the request. An interest for a service object travels towards the object where the specified function is invoked. Finally, *combined content and service objects* contain content data, the default read function to access the data, and additional functions that can be applied on the content. The advantage of such combined objects is that data can be processed directly on the node that holds the object (90). An interest for a combined content and service object must use the content data name. Therefore, interests are always forwarded in direction of the content. The service to be invoked on the data is indicated in an additional name component.

Composed Services

To compose or chain services, Braun et al. (90) proposes to use a *routing header*. Essentially, a routing header is a list of object names that are visited and invoked one after the other. If services require input parameters, they can also be appended as a list to the routing header. The service to invoke next consumes as many parameters as needed from the top of the list. The result replaces the consumed parameters. Thus, it is possible that the output of a service is the input for another service. The routing header can be visited in parallel, but only if services are independent of each other.

Server Selection

To avoid multiple invocations of a service request, servers should first reply with a data packet, indicating their willingness to process the request. SCN suggests initiating a *unicast session* during this packet exchange such that consecutive interests can reach the same service provider. Moreover, the server selection process should consider the distance between server and client as well as the distance between server and required data. Due to this additional information, the authors expect improved server selections, e.g. to choose a service provider close to data if data is large. Gasparyan et al. (94) refined the support for sessions in SCN and described a concrete approach. With a 2-way handshake, a *unique session identifier* is negotiated. Subsequently, the session identifier is used to create long-lasting FIB entries in intermediate nodes which can be used for future forwarding decisions. This enables to reach the same server multiple times, i.e. a session-like bilateral communication between service consumer and provider. Another approach to server selection takes Services over Content Centric Routing (SoCCeR) (95) that does not establish sessions but uses an algorithm to rank forwarding faces specifically for each request and node. Each node forwards a request towards the "best" node according to the ranking. However, SoCCeR is implemented as a control layer on top of the network layer that manipulates underlaying FIBs.

Stateful Services

In order that a shared state is accessed only under controlled conditions, SCN proposes to use sessions, too. They can be established with the same mechanism as used for server selection. During a session, involved players make sure that the state is not accessed in an unintended way. Moreover, the session concept enables multiple clients to connect to the same service and to share the same state.

A disadvantage of SCN compared to SML, RICE and NFaaS is that it cannot distinguish service invocations and content requests due to the uniform naming scheme. Hence, a request for a service object always induces the invocation of the service. There is no possibility to request the service object itself. Code mobility, however, is one of the major advantages of the aforementioned

approaches and essential in IoT and edge computing scenarios. SCN only mentions that services can be deployed on routers with many service requests. Moreover, the lacking differentiation is a problem for setting appropriate PIT timers (see section 5.2/PIT Timings). Nevertheless, it would be possible to approach this problem during session initialization.

7.4 HTTP Caching & RESTful Web Services

HTTP caching considerations and RESTful web services[37] are not that closely related to the ECS as previously discussed works in this chapter. Starting with that the HTTP is allocated on the application layer through to that RESTful web services aim at specific network devices (96) while the ECS is integrated in the network layer and targets multiple nodes to evaluate an expression. In fact, RESTful web services are often implemented by making use of the HTTP. Therefore, they even lie on a separate layer above the HTTP. Moreover, HTTP caching takes place at the proxy and client level, e.g. in web browsers, and not in caches at the network level such as content stores. However, RESTful web services are serverless and support caching. Serverless does not mean that there are no servers or no state. It means that the "application state lives independent of function instance" (97). Hence, no client-specific context is stored on a server between consecutive requests. All information that is necessary to evaluate a request must be present in the request and the client is responsible of keeping states. Obviously, HTTP requests fulfill statelessness because they are limited to stateless HTTP methods. Nevertheless, REST and the HTTP are approaches to request dynamic contents and there are interesting parallels and considerations that are worth mentioning.

Safe and Idempotent Methods

The HTTP knows the concept of *safe methods*. These are methods that "should not[38] have the significance of taking an action other than retrieval" (98). In other words, safe methods should not change any state, i.e. they should be free of side effects (~pure). Accordingly, unsafe methods have side effects (~impure). Methods that should be safe are HTTP GET, HEAD, OPTIONS, and TRACE (5). Obviously, all safe methods are also idempotent. Additionally, PUT and DELETE are also meant to be idempotent because putting or deleting the same element multiple times should have the same effect as a single request (99).

Cacheable Methods

However, HTTP's concept for what is cacheable and what is not cacheable differs from the concept promoted by the ECS. The HTTP generally bases on the assumption of mutable data, e.g. a newspaper website that changes frequently. Truly deterministic results are the exception. Concerning the HTTP, the pivotal questions for caching are if the method is safe and if it does not depend on a current or authoritative response (100). These criteria generally apply to GET, HEAD, and POST. It is comprehensible that GET and HEAD requests are cacheable because they do not have side effects and responses only change if requested resources have changed. Caching responses of POST

[37] Basically, REST is a set of constraints for web services to manipulate web resources using only stateless methods. Web services that are implemented in accordance to REST are interoperable over the Internet.

[38] The authors lie the emphasis on that a requester cannot guarantee a correct (side effect free) implementation of the requested method.

requests is not widely used (101). Moreover, they are only cacheable if they include explicit freshness information, even though they are unsafe. A cached response with explicit freshness information can be seen as an acknowledgment that the POST request has been successfully processed, preventing multiple executions of the request. Although not identical, it therefore has parallels to our transient acknowledgments (tACKs). Table 7.1 summarizes HTTP methods properties.

Table 7.1 – Safe, Idempotent and Cacheable HTTP Methods

HTTP method	Safe	Idempotent	Cacheable
GET	✓	✓	✓
HEAD	✓	✓	✓
POST	✗	✗	✓
PUT	✗	✓	✗
DELETE	✗	✓	✗
CONNECT	✗	✗	✗
OPTIONS	✓	✓	✗
TRACE	✓	✓	✗
PATCH	✗	✗	✗

Cache Control

In the HTTP, considering a response as cacheable does not imply that it cannot get stale or lose its validity. Besides freshness and a related expiration mechanism that play an important role in the HTTP, validity of cached nearly-static and dynamic contents is further controlled over *cache control directives*. Generally, the validation mechanism ensures that no data is uselessly transmitted between cache and server if data has not changed (102). In the following, four interesting cache control directives are explained. Note that all of them appear in server responses. Hence, it is a possibility for producers to control what should happen with their content in intermediate caches:

- **no-cache:** Responses can be cached but caches must ask the original provider for validation before releasing the cached copy. Validation must be asked for every new request and no matter if the cached response is still fresh or not (103). The directive's name can be misleading. It may come from the fact that every request gets through to the provider. Nevertheless, upon validation, the content can indeed be delivered out of a cache.
- **no-store:** In contrast, the no-store directive advices caches not to cache server responses at all. If the resource is requested again, the request must get through to the server and a full response will be returned (104). Hence, this directive is similar to all opaque ECS tags/classes that must not be cached permanently, i.e. `[eph]`, `[eph,aggr]`, `[com]`, and `[prb]` (see Table 5.9).
- **must-revalidate:** As long as a resource is fresh, caches can serve requests without asking the server for validation as it was the case for no-cache. However, as soon as the resource gets stale, caches must validate the resource before continuing to use it (105). A validation mechanism is not integrated in our approach. Nevertheless, together with expiration information, the must-revalidate directive is an opportunity to enhance the

implementation of `[eph,aggr]` expressions, especially if their results do not change frequently or if the worst-case minimum caching period is hard to estimate.

- **immutable:** A provider that includes this directive in a response indicates that the "response will not change during the freshness lifetime of the response" (106). Hence, explicit revalidation requests from users should be ignored and satisfied by the cached resource if it is still fresh. However, immutable responses that got stale must be treated the same way as non-immutable responses. This contrasts with referential transparency and the general understanding of immutability. CCN/NDN data packets as well as results of `[syn]` and `[sem,X]` expressions cannot get stale or lose their validity.

REST Caching

HTTP methods that are considered cacheable can play a role in REST, too. Amongst other things, REST demands clear rules if a response can be cached or not, mainly to avoid stale results but also to facilitate scalability of web services. If these rules are missing, the web service is not considered RESTful. Therefore, web services that want to be RESTful often bases on HTTP and simply define GET and HEAD methods as cacheable and all other methods as non-cacheable. However, the understanding of caching in HTTP-based RESTful web services is rather tolerant. To cache a response, it is enough that a method itself does not *instantaneously* change the representation of a resource, i.e. such that the resource is immediately stale. Other side effects and later changes to the resource are allowed and no pretense not to cache a response. According to Fielding, responses can be cached and reused to satisfy "later requests that are equivalent and *likely* to result in a response identical to that in the cache if the request were to be forwarded to the server" (107). Potentially, again HTTP's validation mechanism should avoid results deviating too much from up-to-date (fresh) server responses. The tradeoff of decreased reliability through stale data is an accepted trade-off in REST (107), but not in the ECS.

Prefetching

Moreover, safe methods are also used for further performance improvements. Results of safe methods can be prefetched. Prefetching is nothing else than a (parallel) out of order method invocation. This important aspect of safe methods can be found again in the ephemeral expression class and is therefore reflected in the ECS.

8. Discussion & Outlook

This chapter critically questions the proposed approach of attributing equivalence classes to differentiate between referentially transparent and referentially opaque expressions, as well as the solution for making interests and data packets request-specific. Moreover, the chapter addresses further theoretical and technical points that have not yet been talked about. Out of it, we motivate future work and derive directions in that the approach can be refined. The end of this chapter concludes the thesis.

8.1 Binding Names to Ephemeral Return Values

So far, we argued with technical reasons that require making expressions with ephemeral return values or problematic side effects distinguishable. For example, packets can get lost or two requests for the same opaque function may appear at the same time. We have seen that content alone is not specific enough to create unique results for referentially opaque expressions. Therefore, a random element that is sufficiently discriminative for a certain time, i.e. until the result has reached its requester, is included in interests and data packets: the nonce. However, the matter has also a security component. Imagine interests $i_{8.1}$ and $i_{8.2}$ that will be satisfied by data packets $d_{8.1}$ and $d_{8.2}$ respectively. The value after the colon sign ':' is the payload.

```
i8.1: /GetRandom(0,100)[eph]{n_0x12345678}
d8.1: /GetRandom(0,100)[eph]{n_0x12345678} : 82

i8.2: /GetRandom(0,100)[eph]{n_0x98765432}
d8.2: /GetRandom(0,100)[eph]{n_0x98765432} : 88

i8.3: /GetRandom(0,100)[eph]{n_0x12345678}
d8.1: /GetRandom(0,100)[eph]{n_0x12345678} : 82 ← ⚠
```

So far, so good. Nevertheless, a requester cannot assume that every intermediate node drops $d_{8.1}$ and $d_{8.2}$ out of its cache shortly after returning them. An attacker can cache data packets forever. Hence, if a requester reuses a nonce by chance ($i_{8.3}$), an attacker can satisfy the request with the old data packet $d_{8.1}$. The requester would think that its request was executed, and that the random number is new. However, this is not the case. The problem is that the combination of name and nonce was not securely unique on a global scale. The random discriminative element must allow so many permutations that either an incidental reuse is highly unlikely or an attacker is unable to store the amount of different answers. We argue that this is feasible, yet not free. For comparison, NDN uses a 32 bit long nonce to detect duplicate interests, enabling to express roughly 4 billion permutations. This is far too few for highly frequented opaque functions, not least because collisions occur way before 4 billion requests. After 20'000 requests, there is already a collision probability of

~4.5%[39]. On the other hand, 267 bits (=33.375 bytes) enable more permutations than atoms in the universe[40]. Assuming a maximum packet size of 8800 bytes, this is ~0.38% of the packet. This is not neglectable, especially if packets are smaller than the maximum packet size. However, the price to pay is appropriate in order to get securely unique results. We leave it an open question what a good nonce length would be. Alternatively, a session can be established to negotiate a secure token between requester and the entity that possesses the mutable state, e.g. like Król et al. (87) are proposing it for client and provider. This induces a notable overhead, especially if many mutable states in different locations are involved. Nevertheless, it guarantees functional correctness and to receive latest states directly from the possessing entity.

Provenance & Integrity of Mutable State Copies

Nevertheless, there are three positive properties of the approach. 1. Value and length of the nonce are chosen by the requester. Hence, if a requester is convinced to have chosen a securely unique nonce, the blame cannot be put on someone else in case of problems. 2. In contrast to malicious caching, malicious PIT aggregation is less of a problem. An intermediate node that does not forward interests will simply be excluded from forwarding decisions and alternative paths will be used. 3. An attacker cannot actively create malicious content that would be accepted by a requester because the attacker is not able to falsify a signature. Accordingly, data *provenance* is assured. However, data *integrity* can be discussed. Data is of integrity insofar it does not contain a virus or the like because the packet was produces by a trusted provider. However, a result that was produced for another request can lead to serious problems and unexpected outcomes, too. Hence, one may argue that an old result is also not of integrity.

8.2 Publishing Signatures

The way we described FIB matching considers only identifiers, i.e. the content name in case of static contents and the routine identifier in case of routines. Accordingly, there was no reason to store more information in the FIB than identifiers and forwarding interfaces. However, having a system where every interest is considered a routine call, FIB matching can be improved. So far, the FIB as well as every other data structure does not recognize if a routine is called with a wrong number of arguments. Instead of implementing an additional directory service to *register* and *lookup* specifications of routines, the FIB can be developed in direction of a *distributed signature service*. Routine signatures can be published as every other content object on the network level. Apart from the routine identifier, the expected number of arguments can be defined over generic parameter names. At the same time, these generic parameter names can be used to give additional hints about the expected information at a certain parameter position. Likewise, the ECS tag can be appended. Name $n_{8.1}$ gives a generic example of a signature that can be used to announce a routine. Name $n_{8.2}$ is an explicit example.

```
n8.1: /A/Routine/Name(placeholder1,placeholder2)[ECS_tag]
n8.2: /Math/Log(base,value)[syn]
```

[39] https://www.wolframalpha.com/input/?i=1-(2%5E32)!%2F((((2%5E32)-(20000))!)*(2%5E32)%5E(20000))
[40] https://www.wolframalpha.com/input/?i=2%5E267

FIBs can parse such announcements and decompose them into routine identifier, number (and name) of arguments and ECS tag. Requesters do not need to download signatures prior to a call. However, if a request arrives at an intermediate node that holds the full signature information, it can check the request for its prospect of success. If the request meets the requirements, it is forwarded. Otherwise, it can immediately be rejected. When using NACKs, the requester can be informed about the reason, i.e. a wrong number of arguments, possibly together with the correct routine signature. The advantage is that only requests with appropriate structure should reach providers. Eventually, some subexpressions have already been resolved before the erroneous form of the request was detected. Consequently, signatures should be checked as early as possible. Nevertheless, already computed subexpressions may be reused by a corrected request.

Note that nodes and their FIBs are not obliged to store complete signatures and to enforce full signature checking in order to unburden providers or requesters. They can still store identifiers only and apply longest prefix matching. However, if there is room, the FIB can offer this service for early detection of interests that are doomed to fail.

8.3 Typed Results

A very natural extension to the publishing and checking of signatures are typed expressions and results. Data types may limit the flexibility of how to resolve expressions, but they bear potential for further improvements. For example, knowing required and used types of argument expressions enables type checking *before* expressions leave requesters. A compiler can check for potential type errors. Type checking is beneficial especially for computations that are distributed over a network because type checking should have a much lower time complexity than communication. Hence, it is a good idea to avoid as many run-time exceptions as possible. Nevertheless, it is still possible to receive wrongly typed results at run-time because a requester ultimately cannot influence the type of a result. It is only possible to gently ask for a certain type.

In fact, data types can be used in both directions. On the one hand, requesters can append data types to expressions in order to inform providers about the desired representation of results. Even if the default return type of a routine is different from the requested type, a provider may have the ability to cast the type of the result in order to meet the requester's wish. On the other hand, providers can use data types in signature publications to announce data types of prospective results. Future work needs to fathom which approach is more reasonable.

Seamless Integration

The advantage of our flexible and extensible attribution system is that data types can seamlessly be integrated. They can simply be appended to the list of attributes. As mentioned above, they can be integrated in requests, such as in $i_{8.4}$ and $i_{8.5}$ or in signature publication like in names $n_{8.3}$ and $n_{8.4}$[41].

[41] Without any further specifications we use the following abbreviations for data types: double = 64 bit floating-point value, string = sequence of UTF-16 code units, int = signed 32 bit integer, bool = Boolean value, short = signed 16 bit integer. This serves only as an example and must not be understood as mandatory specification.

```
i8.4: /Math/Log(2,1024)[syn,double]
i8.5: /Bobs/Interesting/Article[syn,string]
n8.3: /CmRDT/Replica/Increment(argExpr1[int])[com,bool]
n8.4: /My/Home/AvgTemp()[eph,aggr,short]
```

8.4 Feasibility

Of course, the feasibility of such a radically new networking approach must be scrutinized. This section briefly takes up a few of the most urgent questions.

Parsing Effort & Execution Model

Obviously, attributed expressions entail work-intensive parsing to capture all different elements such as identifiers, argument expressions, application spheres, equivalence class tags and further attributes. In terms of network timings, this may lead to problems. Hence, when simply requesting static content, the approach may be inferior to established approaches like CCN and NDN. Nevertheless, we argue that parsing effort is acceptable in terms of application timescale. TLV encoding is known to be efficient concerning parsing and can easily be applied to attributed expressions. Moreover, our attribution mechanism facilitates the differentiation of network timescale and application timescale: composed expressions with an overall equivalence class equal to `[syn]`, e.g. explicitly indicated through wrapping, can immediately be taken out of the laborious extended execution model cycle and treated like ordinary CCN/NDN interests.

What also stands out concerning the extended execution model is that matchings are often more work intensive because not only names/identifiers must be matched, but also nonce values. Additionally, the aggregation of equivalent expressions can have a higher memory complexity. A PIT does not only store an additional arrival interface, but also the unreduced expression. Likewise, the content store should cache known equivalences. However, this effort is well invested if a later request can be satisfied without re-computation. Re-packing and re-signing within the network are also new and therefore additional loads. However, once done, they have a reduced memory complexity than individually stored content objects. Hence, it is a question of trading drawbacks against advantages.

Vain Hope for Aggregation

Although `[sem,X]` expressions can perform efficient computations and rewritings directly on names, and therefore have their right to exist, it may be vain hope to believe that this way of determining equivalences can contribute to aggregate different requests. Maggs and Sitaraman (108) state that "on a typical CDN server cluster serving web traffic over two days, 74% of the roughly 400 million objects in cache were accessed only once and 90% were accessed less than four times". If content is rarely requested twice, it is very unlikely that these two requests for the same content will appear at the same time such that they aggregate. Nevertheless, this statement applies to `[syn]` expressions, too, and therefore on interest aggregation in general. As already mentioned, Dabirmoghaddam et al. (83) come to the same conclusion and speak against aggregation in general. Moreover, the statement of Maggs and Sitaraman that 74% of all results will never be accessed again even challenges if results should be cached at all. However, computations can be much more expensive than content. Hence, it may be worth caching them, even if they are requested rarely.

Cache Filtering

More positively, we can add the argument that if an aggregation occurred, we should quite surely cache the result. Cache filtering mechanisms often work with Bloom filters. A content that appears for the first time only leaves a fingerprint in the filter but is not cached. Only after the fingerprint is already present in the filter, i.e. when the content appears a second time, the content gets cached. Accordingly, if an aggregation was observed, the corresponding result should be cached.

Leaking Information

A reason for this phenomenon is for sure that a lot of requests are personalized or encrypted. Rule sets that should be applied to `[sem,X]` expressions require that concerned expression parts are unencrypted. It is not feasible that authorized intermediate nodes decrypt a request just to check whether there is a rule set to apply or not. Accordingly, `[sem,X]` expressions cannot be used together with self-certifying names. Moreover, the ECS tag itself must be excluded from encryption. Otherwise, the system cannot consult them in order to make wise decisions. Moreover, ECS tags must be perceived as *meta data*. They do not directly enable to draw conclusions about a requester. However, they certainly allow to draw conclusions about contents. For example, it is visible if the content is static, an ephemeral result, or if side effects have been involved. It is an open question how much additional attack surface is introduced by our attribution mechanism.

8.5 Conclusions

In the beginning of this thesis stood the wish to have a better support for referentially opaque expressions and mutable states in NFN. So far, NFN was not clear about if and how it would handle such expressions, mainly because it could not distinguish the expression classes. We deliver solutions for both problems, namely an attribution mechanism for expressions and an according execution model.

Inspired by the idea that names should be matched more flexible against referents, we started to thoroughly reason about equivalences between expressions. The idea to adapt mutual expression matching in dependence of expression properties, e.g. if they can be aggregated or evaluated in parallel, resulted in a comprehensive set of equivalence classes. The classes are inspired by common knowledge about distributed concurrent computing as well as by findings about conflict-free operations. Making use of explicit attributes reduces the problem of distinguishing and handling expressions with different characteristics to a matching problem, i.e. a context-dependent determination of equivalences between attributed expressions. Because equivalence classes depend upon universal properties, the approach is not bound to a specific elaboration like NFN.

The next step was to harmonize theoretical findings with an architecture that relies on uniquely named and immutable data packets. It turned out that more flexible expression matching can be solved without particular difficulties for referentially transparent expressions. However, the architectural requirements do obviously not fit well together with referentially opaque expressions and mutable states. To practically apply the determination of equivalences to referentially opaque expressions, we decided to make interest and data packets sufficiently unique by adding another discriminative element to both packet types, i.e. the nonce. This turned out to be an effective medium

to distinguish homonymous interests on a per-request basis. Adapting our approach to given architectural constraints resulted in an extended execution model. The model summarizes how attributed equivalence classes influence specific protocol steps for interest and data packets, i.e. how expressions must be resolved, aggregated and forwarded, as well as how results are re-packed, cached and returned.

We conclude that the introduction of equivalence classes is an essential step to circumvent major accidents that can occur when working with referentially opaque expressions. Following a few basic rules results in a system that acts according to expectations. Nevertheless, we also critically pointed out that especially the improved aggregation capabilities for referentially transparent expressions will have a tough act to follow in large-scale and task-heterogenous scenarios. Moreover, attributes do not harmonize with self-certifying names. Additionally, they inevitably leak information about requested contents. These two reasons indicate that our proposed approach may be applied only in trusted infrastructures and for homogenous tasks that heavily rely on the reuse of cached results and where a lot of similar requests occur that eventually will aggregate. However, remembering the edgy smart home application, a positive view of the situation is that edge computing scenarios may take place in trusted edge networks only. Our approach is feasible in such scenarios by all means.

Moreover, we believe that our approach captivates due to its generality and extensibility. We attached great importance to create a flexible attribution design. The same way how we attribute equivalence classes to expressions can be used to attribute further individual and expression-specific attributes. Furthermore, our notation rules allow to distinguish between requests for static content and requests for computations. This enables more sophisticated PIT timings. Additionally, equivalence class attributes help to improve CS timings, too, resulting in a more efficient garbage collection of results that are no longer needed.

The thesis clearly showed that referential opacity needs to be considered very carefully. We believe that we have raised the awareness for that topic and that we have given valuable insights. Overall, the proposed solutions and concepts are a significant contribution towards name-based distributed computations in information-centric networks.

9. Appendix

9.1 Source Code Examples

Code Listing 9.1 – Repeated Rounding Problem, Part 1/4 [C99]

```
#include <stdio.h>

int main()
{
   float sum = 0;
   long long steps = 0;
   while(sum < 99999.9f)
   {
      sum += 0.1f;
      steps++;
   }
   printf("Steps: %lli\n", steps);
   printf("Decimal: %.10f\n", sum);
   return 0;
}
```

Code Listing 9.2 – Repeated Rounding Problem, Part 2/4 [C# 7.3]

```
public class Rounding
{
   public static void Main()
   {
      float sum = 0;
      long steps = 0;
      while (sum < 99999.9f)
      {
            sum += 0.1f;
            steps++;
      }
      System.Console.WriteLine("Steps: " + steps);
      System.Console.WriteLine("Decimal: {0:F10}",
            sum);
   }
}
```

Code Listing 9.3 – Repeated Rounding Problem, Part 3/4 [Java 8u171]

```
1  public class rounding
2  {
3     public static void main(String[] args)
4     {
5        float sum = 0;
6        long steps = 0;
7        while(sum < 99999.9f)
8        {
9              sum += 0.1f;
10             steps++;
11       }
12       System.out.println("Steps: " + steps);
13       System.out.println("Decimal: " +
               String.format("%.10f", sum));
14    }
15 }
```

Code Listing 9.4 – Repeated Rounding Problem, Part 4/4 [Python 2.7.14]

```
1  #!/usr/bin/python
2
3  sum = float(0.0)
4  steps = long(0)
5  while(sum < float(99999.9)):
6     sum += float(0.1)
7     steps += 1
8
9  print "Steps: %.lu" % long(steps)
10 print "Decimal: %.10f" % float(sum)
```

Bibliography

1. **Jacobson, Van, et al.** Networking Named Content. *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. New York, NY, USA : ACM, 2009. pp. 1-12. 978-1-60558-636-6.

2. **Zhang, Lixia, et al.** Named Data Networking. *SIGCOMM Comput. Commun. Rev.* New York, NY, USA : ACM, July 2014. Vol. 44, 3, pp. 66-73. 0146-4833.

3. **Tschudin, Christian and Sifalakis, Manolis.** Named Functions and Cached Computations. *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. Las Vegas, NV, USA : IEEE, 2014. pp. 851-857. 978-1-4799-2355-7.

4. **Zhang, Lixia, et al.** NDN Protocol Design Principles. *Named Data Networking*. [Online] [Cited: January 25, 2018.] https://named-data.net/project/ndn-design-principles/.

5. **Fielding, R. and Reschke, J.** RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content / Safe Methods. *IETF Tools*. [Online] June 2014. [Cited: January 3, 2019.] https://tools.ietf.org/html/rfc7231#section-4.2.1.

6. **Internation Organization for Standardization (ISO), International Electrotechnical Commission (IEC).** Information technology -- Open Systems Interconnection -- Basic Reference Model: The Basic Model. *International Organization for Standardization*. [Online] November 15, 1994. [Cited: May 23, 2018.] http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip. ISO/IEC 7498-1:1994(E).

7. **Kurose, James F. and Ross, Keith W.** *Computer Networking - A Top-Down Approach*. Fifth Edition. Boston : Pearson Addison-Wesley, 2010. pp. 74-80. 978-0-13-136548-3.

8. **Braden, R.** RFC 1122 - Requirements for Internet Hosts - Communication Layers. *IETF Tools*. [Online] Internet Engineering Task Force, October 1989. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc1122.

9. **Alexander, S. and Droms, R.** RFC 2132 - DHCP Options and BOOTP Vendor Extensions. *IETF Tools*. [Online] Silicon Graphics, Inc. & Bucknell University, March 1997. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc2132.

10. **Postel, Jon.** RFC 791 - Internet Protocol. *IETF Tools*. [Online] University of Southern California, September 1981. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc791.

11. **Deering, S. and Hinden, R.** RFC 8200 - Internet Protocol, Version 6 (IPv6) Specification. *IETF Tools*. [Online] Check Point Software, July 2017. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc8200.

12. **Senie, D.** RFC 2644 - Changing the Default for Directed Broadcasts in Routers. *IETF Tools*. [Online] Amaranth Networks Inc., August 1999. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc2644.

13. **Partridge, C., Mendez, T. and Milliken, W.** RFC 1546 - Host Anycasting Service. *IETF Tools*. [Online] BBN, November 1993. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc1546.

14. **Deering, S.** RFC 1112 - Host Extensions for IP Multicasting. *IETF Tools*. [Online] Stanford University, August 1989. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc1112.

15. **Imielinski, T. and Navas, J.** RFC 2009 - GPS-Based Addressing and Routing. *IETF Tools*. [Online] Rutgers University, November 1996. [Cited: May 23, 2018.] https://tools.ietf.org/html/rfc2009.

16. **Koponen, Teemu, et al.** A Data-oriented (and Beyond) Network Architecture. *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York, NY, USA : ACM, 2007. pp. 181-192.

17. **Cheriton, D. R. and Gritter, M.** TRIAD: A New Next-Generation Internet Architecture. 2000.

18. **Niebert, N., et al.** The way 4WARD to the creation of a future internet. *2008 IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications*. Cannes, France : IEEE, New York, NY, USA, Sptember 15-18, 2008. pp. 1-5. 978-1-4244-2643-0.

19. **García, G., et al.** COMET: Content mediator architecture for content-aware networks. *2011 Future Network Mobile Summit*. Warsaw, Poland : IEEE, New York, NY, USA, June 15-17, 2011. pp. 1-8. 978-1-905824-25-0.

20. **Xylomenos, George, et al.** A Survey of Information-Centric Networking Research. *IEEE Communications Surveys & Tutorials*. s.l. : IEEE, Q2 2014. Vol. 16, 2, pp. 1024-1049. 1553-877X.

21. **Al-Naday, Mays F., Thomos, Nikolaos and Reed, Martin J.** Information-Centric Multilayer Networking: Improving Performance Through an ICN/WDM Architecture. *IEEE/ACM Transactions on Networking*. s.l. : IEEE, February 2017. Vol. 25, 1, pp. 83-97. 1558-2566.

22. **D'Ambrosio, Matteo, et al.** MDHT: A Hierarchical Name Resolution Service for Information-centric Networks. *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking*. New York, NY, USA : ACM, 2011. pp. 7-12. 978-1-4503-0801-4.

23. **Solis, Igancio.** CCNx 1.0 Changes from 0.x. *IETF 90*. Toronto, Ontario, Canada : s.n., July 2014.

24. **Compagno, Alberto, et al.** To NACK or Not to NACK? Negative Acknowledgments in Information-Centric Networking. *2015 24th International Conference on Computer Communication and Networks (ICCCN)*. Las Vegas, NV, USA : IEEE, August 3-6, 2015. pp. 1-10. 978-1-4799-9964-4.

25. **Psaras, Ioannis, Chai, Wei Koong and Pavlou, George.** Probabilistic In-network Caching for Information-centric Networks. *Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking*. New York, NY, USA : ACM, 2012. pp. 55-60. 978-1-4503-1479-4.

26. **Chai, Wei Koong, et al.** Cache "Less for More" in Information-Centric Networks. [ed.] Robert Bestak, et al. *NETWORKING 2012. Lecture Notes in Computer*

Science. Berlin, Heidelberg, Germany : Springer Berlin Heidelberg, 2012. Vol. 7289, pp. 27-40. 978-3-642-30045-5.

27. **International Telecommunication Union.** Global and Regional ICT Data. *ITU Statistics*. [Online] July 11, 2017. [Cited: June 5, 2018.] https://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2017/ITU_Key_2005-2017_ICT_data.xls.

28. **Laganier, J. and Eggert, L.** RFC 5204 - Host Identity Protocol (HIP) Rendezvous Extension. *IETF Tools*. [Online] April 2008. [Cited: June 8, 2018.] https://tools.ietf.org/html/rfc5204.

29. **Afanasyev, Alexander, et al.** SNAMP: Secure namespace mapping to scale NDN forwarding. *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Hong Kong, China : IEEE, 2015. pp. 281-286. 978-1-4673-7131-5.

30. **Hermans, Frederik, Ngai, Edith and Gunningberg, Per.** Mobile Sources in an Information-Centric Network with Hierarchical Names: An Indirection Approach. *7th Swedish National Computer Networking Workshop SNCNW 2011*. Linköping, Sweden : s.n., June 14, 2011.

31. —. Global Source Mobility in the Content-centric Networking Architecture. *Proceedings of the 1st ACM Workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*. Hilton Head, South Carolina, USA : ACM, June 11, 2012. pp. 13-18. 978-1-4503-1291-2.

32. **Zhang, Yu, et al.** A survey of mobility support in Named Data Networking. *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. San Francisco, CA, USA : IEEE, April 10-14, 2016. pp. 83-88. 978-1-4673-9955-5.

33. **C++ reference.** const and volatile type qualifiers. *C++ / C++ language / Declarations*. [Online] Janauary 17, 2018. [Cited: January 25, 2018.] http://en.cppreference.com/w/cpp/language/cv.

34. **Gosling, James, et al.** Chapter 4. Types, Values, and Variables / 4.12.4. final Variables. *The Java Language Specification / Java SE 8 Edition*. [Online] Oracle Inc., February 13, 2015. [Cited: August 7, 2018.] https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.12.4.

35. **Maddock, Chris, et al.** readonly (C# Reference). *Documentations*. [Online] Microsoft, June 6, 2018. [Cited: August 7, 2018.] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly.

36. **Wagner, Bill, et al.** const (C# Reference). *Documentations*. [Online] Microsoft, July 20, 2015. [Cited: January 25, 2018.] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const.

37. **Church, Alonzo and Rosser, John Barkley.** Some properties of conversion. *Transactions of the American Mathematical Society*. s.l. : AMS, 1936. Vol. 39, 3, pp. 472–482.

38. **Sifalakis, Manolis, et al.** An Information Centric Network for Computing the Distribution of Computations. *Proceedings of the 1st ACM Conference on Information-Centric Networking*. New York, NY, USA : ACM, 2014. pp. 137-146. 978-1-4503-3206-4.

39. **Krivine, Jean-Louis.** A Call-by-name Lambda-calculus Machine. *Higher Order Symbol. Comput.* Hingham, MA, USA : Kluwer Academic Publishers, September 2007. Vol. 20, 3, pp. 199-207. 1388-3690.

40. **Crispin, M.** RFC 1730 - Internet Message Access Protocol - Version 4. *IETF Tools.* [Online] University of Washington, December 1994. [Cited: June 14, 2018.] https://tools.ietf.org/html/rfc1730.

41. **Rose, M.** RFC 1081 - Post Office Protocol - Version 3. *IETF Tools.* [Online] TWG, November 1988. [Cited: June 14, 2018.] https://tools.ietf.org/html/rfc1081.

42. **Zhou, Yuezhi, Zhang, Di and Xiong, Naixue.** Post-Cloud Computing Paradigms: A Survey and Comparison. *Tsinghua Science and Technology.* s.l. : Tsinghua University Press, December 2017. Vol. 22, 6, pp. 714-732. 1007-0214.

43. **Gartner, Inc.** Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. *Gartner.* [Online] February 7, 2017. [Cited: June 20, 2018.] https://www.gartner.com/newsroom/id/3598917.

44. **Ericsson AB.** Ericsson Mobility Rebort. *Ericsson.* [Online] November 2015. [Cited: June 20, 2018.] https://www.ericsson.com/assets/local/news/2016/03/ericsson-mobility-report-nov-2015.pdf.

45. **IDC, Inc.** IoT Infographic. *IDC.* [Online] 2015. [Cited: June 20, 2018.] https://www.idc.com/infographics/IoT.

46. **Cisco, Inc.** Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper. *Cisco.* [Online] February 1, 2018. [Cited: June 20, 2018.] https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html. Document ID: 1513879861264127.

47. —. Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. *Cisco.* [Online] September 15, 2017. [Cited: June 20, 2018.] https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html. Document ID: 1465272001663118.

48. —. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. *Cisco.* [Online] 2015. [Cited: June 21, 2018.] https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf.

49. **Bonomi, Flavio.** Connected Vehicles, the Internet of Things, and Fog Computing. *The Eighth ACM International Workshop on VehiculAr Inter-NETworking (VANET 2011).* Las Vegas, NV, USA : s.n., September 23, 2011.

50. **Bonomi, Flavio, et al.** Fog Computing and Its Role in the Internet of Things. *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing.* Helsinki, Finland : ACM, 2012. pp. 13-16. 978-1-4503-1519-7.

51. **Patel, Milan, et al.** Mobile-Edge Computing – Introductory Technical White Paper. *ETSI.* [Online] September 2014. [Cited: June 21, 2018.] https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf.

52. **Schuster, Rolf.** Boosting User Experience by Innovating at the Mobile Network Edge. *ETSI.* [Online] [Cited: June 21, 2018.] http://www.etsi.org/images/files/technologies/MEC_Introduction_slides__SDN_World_Congress_15-10-14.pdf.

53. **Linthicum, David.** Edge computing vs. fog computing: Definitions and enterprise uses. *Cisco.* [Online] [Cited: June 21, 2018.] https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html.

54. **Yu, Wei, et al.** A Survey on the Edge Computing for the Internet of Things. *IEEE Access.* New York, NY, USA : IEEE, 2018. Vol. 6, pp. 6900-6919. 2169-3536.

55. **Oracle, Inc.** Assignment, Arithmetic, and Unary Operators (Language Basics). *Java Documentation.* [Online] [Cited: January 19, 2018.] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html.

56. **C++ reference.** Assignment operators. *C++ / C++ language / Expressions.* [Online] January 14, 2018. [Cited: January 19, 2018.] http://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_direct_assignment.

57. **Wenzel, Maira, et al.** ++ Operator (C# Reference). *Documentations.* [Online] Microsoft, July 20, 2015. [Cited: January 19, 2018.] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/increment-operator.

58. **Oracle, Inc.** 15.7. Evaluation Order. *Java Language Specification.* [Online] [Cited: June 29, 2018.] https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.7.

59. **Microsoft, Inc.** Run-time evaluation of argument lists. *C# 6.0 draft specification.* [Online] [Cited: June 29, 2018.] https://docs.microsoft.com/en-us/dotnet/opbuildpdf/csharp/language-reference/language-specification/toc.pdf?branch=live.

60. **C++ reference.** Order of evaluation. *C / C language / Expressions.* [Online] November 18, 2017. [Cited: June 29, 2018.] https://en.cppreference.com/w/c/language/eval_order.

61. —. Order of evaluation. *C++ / C++ language / Expressions.* [Online] May 9, 2018. [Cited: June 29, 2018.] https://en.cppreference.com/w/cpp/language/eval_order.

62. **Amazon, Inc.** Atomic Counter. *Working with Items in DynamoDB.* [Online] [Cited: July 4, 2018.] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.AtomicCounters.

63. **Shapiro, Marc, et al.** Conflict-Free Replicated Data Types. [ed.] Xavier Défago, Franck Petit and Vincent Villain. *Stabilization, Safety, and Security of Distributed Systems. SSS 2011.* Berlin, Heidleberg : Springer. Lecture Notes in Computer Science, vol 6976, pp. 386-400. 978-3-642-24550-3.

64. **Blaheta, Don.** The Lambda Calculus. *Brown University, Department of Computer Science, CS173 - Programming Languages.* [Online] October 12, 2000. [Cited: June 25, 2018.] http://cs.brown.edu/courses/cs173/2003/Textbook/lc.pdf.

65. **Kohler, Basil, et al.** NFN-scala. *Github.* [Online] University of Basel. [Cited: June 25, 2018.] https://github.com/cn-uofbasel/nfn-scala.

66. **Tschudin, Christian and Sifalakis, Manolis.** Named Functions for Media Delivery Orchestration. *2013 20th International Packet Video Workshop.* San Jose, CA, USA : IEEE, December 12-13, 2013. pp. 1-8. 978-1-4799-2172-0.

67. **The Institute of Electrical and Electronics Engineers, Inc.** IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985.* New York, NY, USA : IEEE, 1985. pp. 1-20. 0-7381-1165-1.

68. —. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008.* New York, NY, USA : IEEE, August 29, 2008. pp. 1-70. 978-0-7381-5752-8.

69. **Villa, Oreste, et al.** Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. *In Proceedings of the Cray User Group Meeting 2009 (CUG 2009).* Atlanta, GA, US : s.n., May 5, 2009. pp. 1-11.

70. **Zhang, Lixia, et al.** NDN Packet Format Specification 0.3 documentation / Interest Packet / MustBeFresh. *Named Data Networking.* [Online] [Cited: July 27, 2018.] https://named-data.net/doc/NDN-packet-spec/current/interest.html#mustbefresh.

71. —. NDN Packet Format Specification 0.3 documentation / Interest Packet / CanBePrefix. *Named Data Networking.* [Online] [Cited: July 27, 2018.] https://named-data.net/doc/NDN-packet-spec/current/interest.html#canbeprefix.

72. —. NDN Packet Format Specification 0.3 documentation / Data Packet / FreshnessPeriod. *Named Data Networking.* [Online] [Cited: July 27, 2018.] https://named-data.net/doc/NDN-packet-spec/current/data.html#freshnessperiod.

73. —. NDN Packet Format Specification 0.3 documentation / Name / Implicit Digest Component. *Named Data Networking.* [Online] [Cited: August 2, 2018.] http://named-data.net/doc/NDN-packet-spec/current/name.html#implicit-digest-component.

74. **Afanasyev, Alexander, et al.** NDN Technical Memo: Naming Conventions. *Named Data Networking.* [Online] July 21, 2014. [Cited: August 2, 2018.] http://named-data.net/wp-content/uploads/2014/08/ndn-tr-22-ndn-memo-naming-conventions.pdf.

75. **Shi, Junxiao.** ndn-cxx - Task #4396: Provide canonical example for latest data retrieval without selectors in real-time applications. *NFD Redmine.* [Online] January 02, 2018. [Cited: July 27, 2018.] https://redmine.named-data.net/issues/4396#note-1.

76. —. NDN Specifications - Feature #4441: Replace MinSuffixComponents and MaxSuffixComponents with CanBePrefix flag. *NFD Redmine.* [Online] January 9, 2018. [Cited: July 29, 2018.] https://redmine.named-data.net/issues/4441.

77. **Zhang, Lixia, et al.** NDN Packet Format Specification 0.3 documentation / Name. *Named Data Networking.* [Online] [Cited: September 6, 2018.] http://named-data.net/doc/NDN-packet-spec/current/name.html#name.

78. **Afanasyev, Alex, Pesavento, Davide and Shi, Junxiao.** NDN Specifications - Name Component Type Assignment. *NFD Redmine.* [Online] August 22, 2018.

[Cited: September 6, 2018.] https://redmine.named-data.net/projects/ndn-tlv/wiki/NameComponentType.

79. **Richardson, Daniel.** Some Undecidable Problems Involving Elementary Functions of a Real Varaible. *Journal of Symbolic Logic.* s.l. : Association for Symbolic Logic, December 1968. Vol. 33, 4, pp. 514-520.

80. **Klyne, G. and Newman, C.** RFC 3339 - Date and Time on the Internet: Timestamps. *IETF Tools.* [Online] Clearswift Copr. and Sun Microsystems, July 2002. [Cited: August 21, 2018.] https://tools.ietf.org/html/rfc3339.

81. **Resnick, P.** RFC 5322 - Internet Message Format / 3.3. Date and Time Specification. *IETF Tools.* [Online] Qualcomm, Inc., October 2008. [Cited: August 21, 2018.] https://tools.ietf.org/html/rfc5322#section-3.3.

82. **Yu, Yingdi, Afanasyev, Alexander and Zhang, Lixia.** Name-Based Access Control. *NDN, Technical Report NDN-0034.* [Online] October 27, 2015. https://named-data.net/wp-content/uploads/2015/11/ndn-0034-nac.pdf.

83. **Dabirmoghaddam, Ali, Dehghan, Mostafa and Garcia-Luna-Aceves, J. J.** Characterizing Interest Aggregation in Content-Centric Networks. *2016 IFIP Networking Conference (IFIP Networking) and Workshops.* Vienna, Austria : IEEE, May 17-19, 2016. pp. 449-457. 978-3-9018-8283-8.

84. **Zhang, Lixia, et al.** NDN Packet Format Specification 0.3 documentation / Interest Packet / Nonce. *Named Data Networking.* [Online] [Cited: October 24, 2018.] http://named-data.net/doc/NDN-packet-spec/current/interest.html#nonce.

85. **Dijkstra, Edsger W.** How do we tell truths that might hurt? *CS655: Programming Languages.* [Online] June 18, 1975. [Cited: December 07, 2018.] http://www.cs.virginia.edu/~evans/cs655-S00/readings/ewd498.html.

86. **Ferg, Stephen R.** Python & Java: a Side-by-Side Comparison. *Internet Archive Wayback Machine.* [Online] February 7, 2004. [Cited: December 5, 2018.] https://web.archive.org/web/20060428151234/http://www.ferg.org/projects/python_java_side-by-side.html#typing.

87. **Król , Michał, et al.** RICE: Remote Method Invocation in ICN. *ICN '18: 5th ACM Conference on Information-Centric Networking (ICN '18).* Boston, MA, USA : ACM, New York, NY, USA, September 21–23, 2018. 978-1-4503-5959-7/18/09.

88. **Król, Michał and Psaras, Ioannis.** NFaaS: Named Function As a Service. *Proceedings of the 4th ACM Conference on Information-Centric Networking (ICN '17).* Berlin, Germany : ACM, New York, NY, USA, September 26-28, 2017. pp. 134-144. 978-1-4503-5122-5.

89. **Madhavapeddy, Anil, et al.** Unikernels: Library Operating Systems for the Cloud. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13).* Houston, Texas, USA : ACM, New York, NY, USA, March 16-20, 2013. pp. 461-472. 978-1-4503-1870-9.

90. **Braun, Torsten, et al.** Service-Centric Networking. *2011 IEEE International Conference on Communications Workshops (ICC).* 2011. pp. 1-6.

91. **Muscariello, Luca, et al.** Cicn. *FD.io.* [Online] March 12, 2018. [Cited: December 30, 2018.] https://wiki.fd.io/view/Cicn.

92. **Mosko, M., Solis, I. and Wood, C.** ICNRG - CCNx Semantics: Names. *IETF Tools.* [Online] PARC, Inc., LinkedIn, University of California Irvine, June 26, 2018. [Cited: December 31, 2018.] https://tools.ietf.org/html/draft-irtf-icnrg-ccnxsemantics-09#section-3.

93. —. ICNRG - CCNx Messages in TLV Format: Name. *IETF Tools.* [Online] PARC, Inc., LinkedIn, University of California Irvine, July 2, 2018. [Cited: December 31, 2018.] https://tools.ietf.org/html/draft-irtf-icnrg-ccnxmessages-08#section-3.6.1.

94. **Gasparyan, Mikael, et al.** Session Support for SCN. *2017 IFIP Networking Conference (IFIP Networking) and Workshops.* Stockholm, Sweden : IEEE, New York, NY, USA, June 12-16, 2017. 978-3-901882-94-4.

95. **Shanbhag, Shashank, et al.** SoCCeR: Services over Content-centric Routing. *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking (ICN '11).* Toronto, Ontario, Canada : ACM, New York, NY, USA, August 19, 2011. pp. 62-67. 978-1-4503-0801-4.

96. **Oran, David.** Design Challenges for Combining Compute and Networking. *IRTF - Computing in the Network (COIN).* [Online] November 9, 2018. [Cited: January 2, 2019.] https://trac.ietf.org/trac/irtf/raw-attachment/wiki/coin/Design%20Challenges%20for%20Combining%20Compute%20and%20Networking.pdf.

97. **Kutscher, Dirk.** Compute-First Networking (CFN). *IRTF - Computing in the Network (COIN).* [Online] November 9, 2018. [Cited: January 2, 2019.] https://trac.ietf.org/trac/irtf/raw-attachment/wiki/coin/ietf-103-kutscher-coin.pdf.

98. **Fielding, R., et al.** RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1 / Safe Methods. *IETF Tools.* [Online] June 1999. [Cited: September 27, 2018.] https://tools.ietf.org/html/rfc2616#section-9.1.1.

99. **Fielding, R. and Reschke, J.** RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content / Idempotent Methods. *IETF Tools.* [Online] June 2014. [Cited: January 3, 2019.] https://tools.ietf.org/html/rfc7231#section-4.2.2.

100. —. RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content / Cacheable Methods. *IETF Tools.* [Online] June 2014. [Cited: September 28, 2018.] https://tools.ietf.org/html/rfc7231#section-4.2.3.

101. —. RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content / POST. *IETF Tools.* [Online] June 2014. [Cited: January 3, 2019.] https://tools.ietf.org/html/rfc7231#section-4.3.3.

102. **Fielding, R., Nottingham, M. and Reschke, J.** RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching / Validation. *IETF Tools.* [Online] June 2014. [Cited: September 28, 2018.] https://tools.ietf.org/html/rfc7234#section-4.3.

103. —. RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching / no-cache. *IETF Tools.* [Online] June 2014. [Cited: January 5, 2019.] https://tools.ietf.org/html/rfc7234#section-5.2.2.2.

104. —. RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching / no-store. *IETF Tools.* [Online] June 2014. [Cited: January 5, 2019.] https://tools.ietf.org/html/rfc7234#section-5.2.2.3.

105. —. RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching / must-revalidate. *IETF Tools*. [Online] June 2014. [Cited: January 5, 2019.] https://tools.ietf.org/html/rfc7234#section-5.2.2.1.

106. **McManus, P.** RFC 8246 - HTTP Immutable Responses. *IETF Tools*. [Online] Mozilla, September 2017. [Cited: January 5, 2019.] https://tools.ietf.org/html/rfc8246.

107. **Fielding, Roy Thomas.** *Architectural Styles and the Design of Network-based Software Architectures*. [Dissertation] s.l. : University of California, Irvine, 2000.

108. **Maggs, Bruce M. and Sitaraman, Ramesh K.** Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*. s.l. : ACM, New York, NY, USA, July 2015. Vol. 45, 3, pp. 52-66. 0146-4833.

109. **Newman, D.** RFC 2647 - Benchmarking Terminology for Firewall Performance. *IETF Tools*. [Online] Data Communications, August 1999. [Cited: May 25, 2018.] https://tools.ietf.org/html/rfc2647#section-3.17.

110. **Amazon Web Services, Inc.** Globale AWS Infrastruktur & Availability Zones. *AWS*. [Online] [Cited: June 19, 2018.] https://aws.amazon.com/de/about-aws/global-infrastructure/.