



Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

How (Not) to Design an Internet: Principles and Properties of Network Architectures

A dissertation by

Dirk Hasselbalch
hasselbd@tcd.ie

submitted to

the University of Dublin
Trinity College

in fulfilment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

June 2017



Thesis 11477

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

2017-06-10
Date



I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Date

Dirk Hasselbalch

Abstract

The TCP/IP internet architecture (IA) has been tremendously successful and the Internet has become a dominating component in the world's economic and political landscapes. However, the Internet is based on a prototype network that outgrew itself before its teething troubles were rectified and it has serious issues with flexibility, mobility, scalability, and security: flexibility is lacking due to a bottom-up intensional design, whereby internal identifiers and semantics are exposed through the APIs; mobility is either not transparent to applications and requires DNS involvement, or it does not scale; scalability is sacrificed for mobility or to avoid network renumbering; and security needs to be provided by applications, alternative protocols, or middleboxes.

Multiple future IA (FIA) proposals as well as numerous augmentations to the classic TCP/IP IA have been devised to remedy these issues. They have very different goals, designs, and properties, which make them difficult to compare meaningfully. Consequently, the research community seems unable to agree on common solutions to the deficiencies.

In this dissertation, we seek to address this situation by providing a qualified assessment of several contemporary and future IAs with a special focus on the classic Internet (IP version 4 (IPv4)-based TCP/IP and DNS), modern Internet (the Internet with its modern augmentations such as IP version 6 (IPv6), Software-Defined Networking (SDN), Mobile IP (MIP), and the Locator/Identifier Separation Protocol (LISP)), the Recursive Internet Architecture (RINA), and Named Data Networking (NDN). To enable such an assessment, we define a taxonomy based on extensional *properties* desired for IAs identified from the FIA literature, along with fundamental and firmly anchored intensional *principles* from software systems design, which at times has been down-played in network and Internet research; instead, the Internet Engineering Task Force (IETF) mantra of 'rough consensus and running code' (Day 2008a) has been emphasised.

Using the taxonomic framework, available literature, analysis of available prototype APIs and code, and first-hand experience, we identify several root causes of the deficiencies

of the IAs, and provide a thorough comparison, resulting in a rating of each IA. We note that most actual IAs do not adhere to the principles identified in the taxonomy and also lack many of its desired properties. A significant finding is that for the IAs assessed, the IAs that adhere to the principles, have more of the desired properties. Based on these observations, we claim that the lack of desirable properties in IA are linked to the lack of adherence to principles, and therefore that a greater focus on these principles is to be desired in IA research. Our results points to RINA as the 'best' IA of the ones assessed given its employment of principles and achieved properties.

Acknowledgements

First of all, I would like to thank my supervisors Eric Jul and Mélanie Bouroche for friendship, for guiding me through the PhD process culminating in this dissertation and for providing me with encouragement and advice. Also thank you to my examiners, Stefan Weber and Boris Magnusson, for feedback during and after my viva; to David Gregg for chairing the viva; and to Mads Haahr for substituting Mélanie during her maternity leave.

Thank you to my managers at Nokia Bell Labs Ireland: Holger Claussen, who agreed to take me on as an industrial PhD student; Alessandra Sala, who has provided day-to-day leadership and friendship; and Volker Hilt, who has encouraged and supported me in the second half of the period.

A special thank you to John Day, who during his visit to Trinity College Dublin in June 2015 took the time to engage in numerous discussions about Internet architecture, and with whom many insightful exchanges over email and video-calls have taken place since then. Thank you for providing extensive feedback on written pieces and thoughts, and for continually keeping me up to date with your own ideas as well as with the most recent research within the RINA community.

Another special thank you to Scott Shenker and Barath Raghavan, for a fruitful collaboration and many interesting discussions on internets in general and the Software-Defined Internet Architecture (SDIA) in particular during my stay at the International Computer Science Institute at UC Berkeley (ICSI) in the fall of 2014, during a short visit there in August 2015, and in numerous video-calls in between. Thanks to Scott Shenker for inviting me, funding the visits, and taking the time for discussions; it has been an utmost privilege and has provided me with invaluable experience, even though the work we did, did not make it into this dissertation. Thank you to the staff at ICSI for their friendliness and for making me feel very welcome during both visits.

Thank you to Fritz Henglein for generously providing me and Eric an office at the Department of Computer Science, University of Copenhagen (DIKU), allowing me to finish

this dissertation from my alma mater.

A big thank you to my friends and family, who in times of trouble have provided guidance and encouraged me to continue on, and especially to Franck Franck for getting me started on the PhD track, for professional companionship and good friendship and for being there to share the ups and downs of PhD studentship in a country away from home.

I would also like to thank my colleagues, fellow academics and friends at Nokia Bell Labs Ireland, in the Distributed Systems Group at the School of Computer Science and Statistics at Trinity College Dublin, and at DIKU for professional feedback, friendship, banter and long lunches.

Last but not least, thanks to Nokia and the Irish Research Council for funding me throughout the period under the Irish Research Council Enterprise Partnership Scheme, Project ID: 200849, Award: 11849.

Contents

Abstract	iii
Acknowledgements	v
Contents	vii
1 Introduction	1
1.1 Motivation	1
1.2 Goals	2
1.3 Approach	3
1.4 Work Done	3
1.5 Contributions	3
1.6 Challenges	4
1.7 Results	4
1.8 Outline and Reader's Guide	4
2 Background	7
2.1 Terms and Definitions	7
2.1.1 Internet Architecture	7
2.1.2 Other Concepts	9
2.2 State of the Art	10
2.2.1 Classical TCP/IP internet architecture (IA) and Legacy IAs	10
2.2.2 Building on the TCP/IP IA	13
2.2.3 Modern Clean Slate IAs	15
2.2.4 Summary and Relation to This Dissertation	16
2.3 Principles	17
2.3.1 Modularity via Abstractions	18
2.3.2 Black Boxes and Extensionality	19
2.4 Protocols and Interfaces	20
2.4.1 Protocols	21
2.4.2 Interfaces	22
2.4.3 Protocol, Services and Extensionality	22
2.4.4 Summary	23
2.5 Location	24
2.6 Properties	25
2.6.1 Mobility	25

7.5.4	Overloaded Pervasive Names	154
7.5.5	Names and Addresses	154
7.5.6	Architectural Issues	155
7.5.7	Lack of System Boundary	155
7.5.8	NDN Explained in RINA Terms	156
7.5.9	Premise of NDN	157
7.5.10	Conclusions	158
7.6	Properties	159
7.6.1	Flexibility	159
7.6.2	Mobility	160
7.6.3	Scalability	163
7.6.4	Security	163
7.6.5	Simplicity	164
7.7	Principles	168
7.7.1	Indirection	169
7.7.2	Encapsulation	171
7.7.3	Extensionality	172
7.7.4	Generality	174
7.8	Summary and Perspective	175
7.8.1	Indirection	175
7.8.2	Encapsulation	176
8	Conclusions	179
8.1	Contributions	180
8.2	Perspective	180
	References	183
	Glossary	191
	Acronyms	193
	List of Figures	201
	List of Tables	205

Chapter 1

Introduction

In this dissertation, we assess several internet architectures (IAs) with a focus on five central properties: flexibility, mobility, scalability, security and simplicity, and their relation to four principles of systems design: extensionality, generalisation, encapsulation and indirection. We have analysed four IAs thoroughly, and we find that most IAs have issues with several of the properties, and that the issues can be traced back to the neglect of one or more of the principles. Conversely, we also find that the IA that have achieved desirable properties have done so through the deliberate application of these principles.

1.1 Motivation

There is no doubt that the Internet is a tremendous success: It is shaping all aspects of our society, and is a driver of the economy of the information age. Although successful, it has serious technical issues with mobility, security, scalability, flexibility and complexity, as we explore in Chapter 4.

To address these problems, several augmentations have been devised for the traditional Internet technology stack, which is centred around TCP/IP, DNS and Border Gateway Protocol (BGP). Examples include IP version 6 (IPv6), Software-Defined Networking (SDN), Network Function Virtualization (NFV), the location/identifier split (loc/id)—including the Locator/Identifier Separation Protocol (LISP), the Host Identifier Protocol (HIP) and the Identifier-Locator Network Protocol (ILNP)—and Mobile IP (MIP), as well as lower layer augmentations, for example Multi-Protocol Label Switching (MPLS) (Srinivasan et al. 2004). They try to maintain as many of the *invariants* of the current architecture as possible so that existing applications and infrastructure need only little adjusting.

In addition to these, several alternative IAs (aka. future IAs (FIAs)) have also been proposed recently, including various information centric networks (ICNs)—for example, content centric networking (CCN) aka. Named Data Networking (NDN) (Jacobson et al. 2009; Zhang, Afanasyev, et al. 2014), the Network of Information (NetInf) (Dannewitz et al. 2013), the Data-Oriented Network Architecture (DONA) (Koponen, Chawla, et al. 2007)—the eXpressible Internet Architecture (XIA), MobilityFirst (MF), and the Software-Defined Internet Architecture (SDIA). Many have further spin-off projects or variations. Common to these is their self-proclaimed *clean slate* nature: They wholly or partly depart from the legacy of TCP/IP and get to choose their invariants themselves.

While there is some agreement on what the symptoms are—lack of mobility and scalability, lack of security, rigidity in choice of addresses and protocols—there is little consensus on the importance of each in relation to the others, on what causes them, or on how they should be solved. This has led to a state of the art where each solution argues for its own merits based on its own premises, but where objective ways to compare solutions are not applied. Thus one solution is as good as the other: Each has different pros and cons, and that we cannot compare oranges and apples, so we should not try.

To break that barrier, we use a taxonomy that puts all IAs on common ground. Our taxonomy is based on fundamental principles from systems design. The principles are centred around modularity and abstractions, and they form the foundation for both the object-oriented design and functional programming paradigms, and there is a general long-standing consensus that these principles lead to better quality systems (D. L. Parnas 1972; D L Parnas et al. 1985; Stevens et al. 1974).

We further observe that IAs specify a way to design a distributed system with the purpose of providing a communication service. We therefore identify desirable properties that any such service should possess and include that in our taxonomy. We then use the taxonomy to assess the current IA and several augmentations and FIAs.

1.2 Goals

Our goal is to provide assessments of IAs that are independent of the premises of the IAs in question, and instead are based on more general premises based on ‘good Internet architecture’. The purpose is to be able to compare the qualities of a few of the IAs in existence to distil a set of good practices that can pave the way forward for the evolution of the Internet.

As such, our aim is to help setting the stage for the next generation Internet by pointing out where a selected set of current IAs fail or succeed and why.

1.3 Approach

Our approach is to devise a taxonomy for assessing IAs based on desirable properties of IAs and fundamental principles from systems design and use this taxonomy to assess several IAs.

Our assessment of individual IAs and augmentations is based on a design and model analysis derived from available literature, standards, and implementations, first hand experience with a subset of the IAs, including TCP/IP, NetInf, DONA, SDIA, and MF, and lengthy conversations and clarifications from the authors of SDIA and the Recursive Internet Architecture (RINA); as well as on a categorisation of each IA in relation to the taxonomy.

1.4 Work Done

To complete this dissertation, we have carried out the following work:

- Survey of existing and future IAs, as well as related subjects such as systems design, protocols, naming, OS'es, and distributed hash tables (DHTs).
- First-hand experience with NetInf, SDIA, MF, DONA, and TCP/IP, and lengthy conversations with the authors of SDIA and RINA.
- Formulation of a taxonomy of principles and properties for IAs based on literature on systems design and IAs.
- Analysis and categorisation of IAs in the taxonomy.

1.5 Contributions

This dissertation provide the following contributions:

1. A case for more discipline and adherence to core principles in network architecture design.

based on:

2. A taxonomy for the assessment of IAs.
3. A survey and analysis of several IAs and augmentations.
4. A categorisation and assessment of these IAs in relation to the taxonomy.

1.6 Challenges

It is a challenge to devise a set of objective parameters with which to evaluate a given system architecture because the quality of a system is, bluntly, the degree to which it meets or exceeds expectations. Thus, the quality of a system depends on the properties of the system compared to the properties expected from its users. Therefore, we do not presume that our evaluation parameters are objective.

However, because IAs are large, distributed software systems, we posit that the principles that has proven valid for distributed systems also applies to IAs. While this should really not be a controversial statement, it does go against the grain of the Internet Engineering Task Force (IETF), which believes in ‘rough consensus and running code’ (Hoffman and Harris 2006) rather than abstract principles.

We argue that the properties of IAs that we identify as being worthwhile indeed are worthwhile, and that there is at least some consensus in the research community that this is the case as various such properties are emphasised in IAs. We hypothesize that the principles that we identify as fundamentally ‘good’ are so because they lead to desirable properties. As such, they are not a priori good or bad, but we posit that using them leads to desired properties in the systems in which they are applied that we have assessed.

1.7 Results

This dissertation has resulted in the following:

1. A taxonomy of principles and properties, which in addition to its use in this dissertation, can be useful for future work evaluating or comparing complex and/or distributed systems.
2. A survey, categorisation and assessment of several contemporary IAs.
3. A conclusion calling for better discipline and adherence to fundamental principles.

1.8 Outline and Reader’s Guide

The remainder of this dissertation have the following structure:

- Background (Chapter 2)

We describe the concepts and terminology used throughout the dissertation. This includes the principles of our taxonomy, namely modularity, top-down versus bottom-up models and back-boxes, encapsulation and indirection; the properties of our taxonomy, namely flexibility, mobility, scalability, security and simplicity; as well as other relevant concepts, namely protocols, interfaces, location, and naming.

- Presentation of taxonomy (Chapter 3)

We define the taxonomy we use to assess the IAs in Chapters 4 to 6 and Section 4.5.

- Presentation, categorisation and assessment of IAs and augmentations (Chapters 4 to 6 and Section 4.5)

We present a range of IAs and augmentations, and for each one, we categorise and assess it according to our taxonomy and discuss our findings.

- Discussion of findings (Chapter 7)

We discuss our findings in more general terms, and compare and assess the IAs.

- Conclusions (Chapter 8)

We conclude and summarise our results.

Reading Guide We promote the following reading guide:

- For the top level message, read Abstract (page iii), Introduction (page 1), and Conclusions (page 179).
- For a high-level read, supplement with Background (page 7), Taxonomy (page 29), and Discussion (page 117), and include only the introduction and summary sections of each of the four IA chapters (i.e., introduction and summary parts of Chapters 4 to 6 and Section 4.5).
- For most details, read the full dissertation.

Chapter 2

Background

In this chapter, we account for fundamental terms and definitions and give a brief presentation of the state-of-the-art with a view that the bulk of the dissertation is in itself a presentation of the state-of-the-art along with a critique of ways to do internet architecture.

We identify some important principles of system design including Modularity via Abstractions and the underlying principles of Indirection, Encapsulation, Generality, and Extensionality.

We discuss the connections between protocols, services, and extensionality.

We describe how the concepts of location, mobility, and security influence the design of an internet architecture and how the principles can be used to provide a basis for incorporating these concepts into what we claim is a sound design.

Finally, we choose five properties to evaluate network architectures.

2.1 Terms and Definitions

We here define the key concepts used throughout the dissertation.

2.1.1 Internet Architecture

We will here define ‘internet architecture’ based on the definition of ‘software architecture’ in Fielding (2000, Chapter 1), which is as follows:

A **software architecture** is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software

architecture.

...

[It] is defined by a configuration of architectural elements—components, connectors, and data—constrained in their relationships in order to achieve a desired set of architectural properties.

Fielding (2000) then continues to define the concepts contained in this definition, in particular the ‘architectural elements’—components, connectors, and data—and ‘configuration’:

A **component** is an abstract unit of software instructions and internal state that provides a transformation of data via its interface.

A **connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components.

A **datum** is an element of information that is transferred from a component, or received by a component, via a connector.

A **configuration** is the structure of architectural relationships among components, connectors, and data during a period of system run-time.

We find no reason to challenge this definition of software architecture. We will therefore use it to define what we mean with internet architecture. To do this however, we first need to define two important concepts: ‘networking software system’ and ‘network architecture’.

A **networking software system** is a software system whose purpose it is to provide inter-process communication (IPC) between components via a computer network.

The fact that networking is based on IPC can be attributed (Day 2008a; Day et al. 2008a) to a quote by Robert Metcalfe, the architect behind IEEE 802.3 (Ethernet):

‘Networking is inter-process communication.’

Importantly, *the* Internet is a networking software system. *An* internet, however, is not equivalent to *the* Internet:

An **internet** is a networking software system that seeks to replace, is able to replace, or is the Internet.

A ‘network architecture’ is then defined as follows:

A **network architecture** is the software architecture of a networking software system.

We define an ‘internet architecture’ as follows:

An **internet architecture** is the software architecture of an internet.

Similarly to the fact that there is a distinction between *the* Internet and *an* internet, there is a distinction between *the* Internet architecture, and *an* internet architecture—the former, we define as follows:

The **Internet architecture** is the software architecture of the Internet.

It follows, then, that the Internet architecture is an internet architecture, which is a network architecture, which is a software architecture.

Throughout this dissertation, we use the abbreviation IA for both the Internet architecture and an internet architecture, where the article (‘the’/‘an’) will define which one.

2.1.2 Other Concepts

Day (2008a) reuses numerous IA-independent terms from Open Systems Interconnection (OSI) (ITU-T 1994), and define several additional ones. The terms have been honed in Day, Bunch, et al. (2013). For this dissertation, we will rely on the following definitions from Day, Bunch, et al. (2013):

Protocol: the syntax of protocol data units (PDUs) and associated set of procedures, which specifies the behaviour between two protocol machines (PMs) for the purpose of maintaining shared state.

Protocol machine (PM): a finite state machine that implements a protocol, that accepts input from its user and exchanges PDUs with a peer by transferring service data units (SDUs) to a supporting service with the intent to maintain shared state with a corresponding PM usually in another processing system (PS).

Protocol data unit (PDU): the unit of data exchanged by PMs.

Protocol control information (PCI): that portion of the PDU, which is interpreted by the PM to maintain shared state of the protocol.

Service data unit (SDU): that portion of the PDU, which is not interpreted and is not interpretable by the PM and is delivered transparently to its client.

Processing system (PS): the hardware and software capable of executing programs instantiated as application processes (APs) that can coordinate with the equivalent of a “test and set” instruction, i.e., the tasks can all atomically reference the same memory.

Application process (AP): the instantiation of a program executing in a PS intended to accomplish some purpose.

Application entity (AE): a task within an AP directly involved with exchanging application information with other APs.

2.2 State of the Art

We here review some of the existing literature on IA research and position our work in relation to it. This is not an exhaustive list of existing literature.

We categorise the existing work we have chosen to include into three main categories:

1. Classical TCP/IP IA and Legacy IAs.
2. Building on the TCP/IP IA.
3. Modern Clean Slate IAs.

2.2.1 Classical TCP/IP IA and Legacy IAs

Before TCP/IP became the de-facto architectural standard for global internetworking, several research efforts were concerned with network protocols and architectures. Apart from the the U.S. Department of Defence’s Advanced Research Projects Agency Network (ARPANET) project (Bolt Beranek and Newman, Inc 1981), the TCP/IP protocol suite and model (V. G. Cerf et al. 1983; Clark 1988), was greatly influenced by the OSI standardisation efforts (Day and Zimmermann 1983) and the Cyclades project (Pouzin 1973). In addition, Day (2008a) provides historical context surrounding these publications. Furthermore, seminal textbooks on networks, such as Tanenbaum and Wetherall (2011) and Kurose and Ross (2013), lays out the basic understanding gained from these efforts aimed at our generation’s students of computer science.

The ARPANET was the world’s first functional packet-switched network according to Bolt Beranek and Newman, Inc (1981), who lays out the first decade of the ARPANET, starting from the conception in the “middle to late 1960s”, over its construction starting in

1969, and until its handover to the U.S. Defense Communications Agency (DCA) in 1975. It describes the basic architecture of ARPANET as follows:

- The ARPANET's hosts are a collection of heterogeneous computers, used for a variety of applications.
- A host is connected to one Interface Message Processor (IMP). An IMP may have several hosts connected to it.
- IMPs are small, special-purpose computers. Their purpose is to route data.
- Telephone lines connect the IMPs. The IMPs and telephone lines makes up the network of the ARPANET.

This is depicted in Figure 2.1.

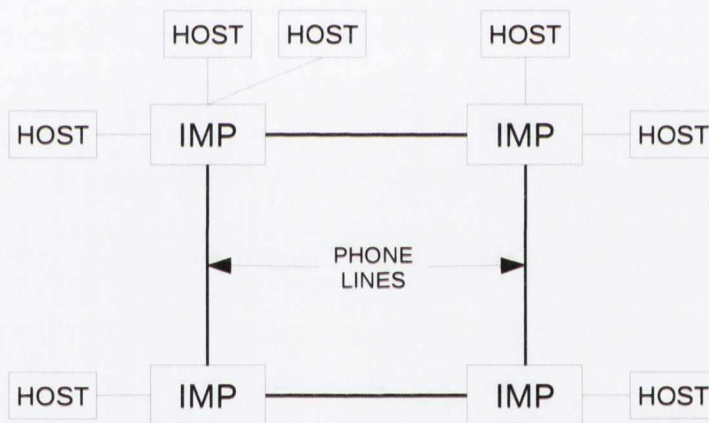


Figure 2.1: The basic ARPANET architecture, consisting of hosts, IMPs, and telephone lines. Adapted from Bolt Beranek and Newman, Inc 1981, Figure 1.

According to Bolt Beranek and Newman, Inc (1981), the ARPANET was greatly influenced by contemporary research on packet switched networks, in particular Baran (1964), Davies et al. (1967), and Pouzin (1973), and was the first to apply these principles in a production network. Baran (1964), according to Day (2008a), was motivated by the idea a resilient network capable of surviving loss of network nodes, which led to the concept of reliable connections running on top of an otherwise unreliable network. On the ARPANET, hosts exchanged longer messages of 8000 bit which was broken into 1000 bit packets by the IMPs. The shorter packages were routed by the IMPs, and the the Network Control Program (NCP) running on hosts ensured primitive reliability. Cyclades (Pouzin 1973) took the idea one step further, and introduced connection-less networks, where individual short datagrams were

exchanged unreliably between hosts without a reliable, connection-oriented abstraction. Day (2008a) argues that because Cyclades was shut down in the 1970’s and no further efforts were made into merging the concepts into a generalised communication abstraction, the connection-less/connection-oriented split was prematurely cemented, ending up as two different protocols in TCP/IP: TCP for the connection-oriented abstraction and IP (with UDP being a thin abstraction layer on top) as the connection-less one.

The concept of network layers was motivated by similar abstractions within OS research, for example the layers of the THE OS (Dijkstra 1968), because the engineers working on ARPANET and the Internet were primarily OS experts (Day 2008a, p. 61). The layered TCP/IP reference model (V. G. Cerf et al. 1983), was the result of practical and theoretical work on ARPANET. It influenced and was influenced by the OSI protocols and standardisation efforts (Day and Zimmermann 1983). It is shown in Figure 2.2.

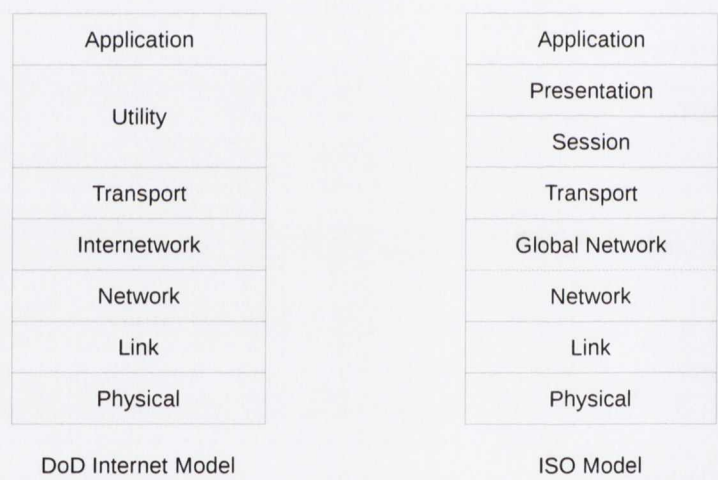


Figure 2.2: The layers of the Internet: TCP/IP reference model layers and OSI layers.
Adapted from V. G. Cerf et al. 1983, Figure 4.

Clark (1988) summarizes the motivation and the design goals of the Internet. The main goal “was to develop an effective technique for multiplexed utilization of existing interconnected networks” (Clark 1988, p. 106, Section 2), amongst which was the existing ARPANET. The fact that existing networks and applications which where to be joined together by the Internet were packet-switched ones made packet switching a fundamental component of the Internet architecture. Another important aspects was the choice to let the various networks be interconnected by packet store-and-forward gateways, which let each network keep its existing network architecture.

2.2.2 Building on the TCP/IP IA

We here describe some of the research that builds on the existing TCP/IP IA.

Fielding (2000) presents the Representational State Transfer (REST) architecture that seeks to improve on the World Wide Web (the Web) architecture, which itself is built on the TCP/IP IA. The TCP/IP IA itself is not evaluated, which makes its scope notably different from this dissertation, which seeks to do exactly that. Fielding presents a terminology for software architecture in general and lists key properties of network-based application architectures in particular. He specifically scopes the work to 'application architectures, excluding the operating system, networking software, and some architectural styles that would only use a network for system support' (Fielding 2000, Section 2.1.2). In contrast, the focus of this dissertation is IAs, where the OS, the networking software, and the interaction between those and between them and the applications are critically important. This difference in scope means that the list of key properties applicable to application architectures he lists (Fielding 2000, Section 2.3) on are not identical to the list of properties applicable to IAs we present in Chapter 3 (although some properties are important in both cases, and therefore exist on both lists). The terminology for software architecture presented by Fielding is valuable. We have, however, not used it in this dissertation. Utilizing those aspects in this work would likely improve the present dissertation, and would be a valuable item for future work.

Popa et al. (2010) explores how the Web can be used as the basis for an IA with the addition of a new method inhabiting subscribe semantics to the existing HTTP protocol with which it will, according to the authors, be able to support both ICN and delay or disruption tolerant networking (DTN) semantics. They conclude that 'HTTP can be used to provide many of the properties proposed by previous architectural proposals'. Compared to this dissertation, they do not analyse the interaction between the components in the current TCP/IP IA and instead focus on the service provided by their proposal compared to that of existing FIA. They do not mention the issues with the current IA that we identify in Chapter 4.

DONA (Koponen, Chawla, et al. 2007) proposes a service that allows its users to find data objects by identifiers. It is meant to run on the current TCP/IP IA. It is one of the seminal papers of ICN, although it was not until CCN (Jacobson et al. 2009) two years later that the ICN field gained scientific traction. NetInf (Dannewitz et al. 2013), like DONA, is an ICN that requires a supporting packet-delivery network, such as the Internet, in order to run. So, both DONA and NetInf are overlays, placing themselves between the packet

delivery service provided by the Internet and applications. They provide applications with abstractions centred around named pieces of data.

An alternative to changing abstractions to data is to change the semantics of the IP address from ‘locator’ (i.e., point of attachment (PoA)) to ‘identifier’ (i.e. host) from the viewpoint of the applications. Because routers still need location semantics to route, the translation between locator IPs to identifier IPs must happen either in hosts, before packets reach the application, or in-network in intermediary gateways close to the destination/source hosts. This line of research has triggered several proposals for what has been termed the location/identifier split (loc/id), for example LISP (Farinacci et al. 2013), HIP (Moskowitz, Heer, et al. 2015; Moskowitz and Nikander 2006), and ILNP (Atkinson and Bhatti 2012).

Both ICN and loc/id depend on the current TCP/IP IA, and both attempt to solve the problem of mobility and the dependence between applications and network location created via the socket API and its exposure of IP. Neither write off TCP/IP; they attempt to improve it by extending the existing architecture. This is in contrast with so-called *clean slate* IAs that specifically do not rely on an existing IA.

Software-Defined Networking (SDN) (Haleplidis et al. 2015) and Network Function Virtualization (NFV) (Chiosi et al. 2012) both enable network management, i.e., controlling the routing and forwarding state of networks running TCP/IP and supporting protocols (such as Ethernet).

SDN defines an architecture for controlling routing state of a network by means of a controller that network administrators can configure and that can react dynamically to network conditions. The controller transfers and manages routing state of each of the routers in the network. This is different from the traditional decentralised routing state and management methods used in standard distributed routing protocols such as Open Shortest Path First (OSPF) (Moy 1998) and Intermediate System to Intermediate System (IS-IS) (International Standardization Organization (ISO) 2002).

While SDN is a framework for managing and propagating routing state in a network, NFV is the application of virtualisation to network nodes such as routers, firewalls, or switches. By virtualising network nodes, upgrades with regards to functionality and new protocols can be carried out faster and cheaper, and network providers can reduce their ‘innovation cycles’ (Chiosi et al. 2012). NFV is complementary to SDN: While the latter facilitates management of routing state, the former facilitates management of forwarding behaviour by moving the forwarding mechanism implementations from hardware to software.

UNIFY (Szabó et al. 2014) leverages SDN and NFV and defines a cloud and orchestration

architecture for TCP/IP.

2.2.3 Modern Clean Slate IAs

CCN/NDN (Jacobson et al. 2009; Zhang, Afanasyev, et al. 2014) question the type of service provided by TCP/IP and consider its ‘host-centric’ paradigm outdated. On this premise, they formulate a new ‘content-centric’ architecture. They do not further substantiate their premise that the popularity of a service should decide whether or not it should also be the fundamental one.

While CCN (Jacobson et al. 2009) ignited the current ICN efforts and had few other architectures to relate to, the important ones there are, including DONA (Koponen, Chawla, et al. 2007), are ignored. NDN (Zhang, Afanasyev, et al. 2014) also ignores other FIA and ICN proposals, despite many having appeared in the time since CCN.

Neither analyse why TCP/IP is lacking, only state that it is. They reuse many of its principles, including its aggregation-based routing mechanisms and algorithms (Hoque et al. 2013; Wang et al. 2012). They do not question whether or not the layered model built around TCP/IP is correct or whether or not it should exist without TCP/IP itself, and instead defines itself in relation to it. APIs are neglected, and there is no black-box specification of the proposed CCN/NDN service as a whole; therefore there is no critique of the TCP/IP APIs either.

XIA (Xylomenos et al. 2014) expands on the premise of ICN to allow for names to identify any ‘principal’, where a principal can be data, routers, networks, applications, or some future definition. It focuses on ‘evolvability’, i.e., the ability of a system to adapt to changing requirements over time (what we in this work call flexibility). It defines an adaptive packet delivery protocol, where sources and destinations are typed as the aforementioned principals. If a principal type is unsupported in some areas of the network, the protocol header contains a directed acyclic graph (DAG) which defines fallback principals by which to route, with the bottom case being the current IP networks.

SDIA Ghodsi et al. 2011b; Koponen, Shenker, et al. 2011; Raghavan, Koponen, Ghodsi, Brajkovic, et al. 2012; Raghavan, Koponen, Ghodsi, Casado, et al. 2012 builds on SDN and NFV and also engages the concept of evolvability, but in a different way: They define an architectural framework with which new IAs can be defined within certain limitations.

Day (2008a) provides a thorough and detailed critique of the contemporary Internet architecture. The author identifies its shortcomings, including the lack of mobility, the

non-scalable routing, the lack of security, and revisits the concept of layers; as well as classic protocol design, the use of identifiers and names, and the APIs.

Using principles from classical complex system design based on abstractions, modularity, and information hiding, a new IA is proposed, RINA, which is a complete network architecture providing global IPC, conceptually similar to TCP/IP in many ways, but different in vital design choices. However, other FIA are not considered, although Day 2011 does provide criticism of the overall FIA efforts.

In Day 2011, Day identifies the problems' root causes and back-tracks through history to find out how they arose, bringing in personal experiences and long-forgotten angles provided by OSI and CYCLADES that, he claims, are not obsolete. He alleges that many of the dogmas that are taken for granted by most are not always based on technical soundness but just as often on who won a certain argument.

2.2.4 Summary and Relation to This Dissertation

The FIA literature is fragmented, which is also noted to be the case at least within ICN by Ghodsi et al. 2011a. Day (2011) notes how most seem to be busy designing new services, without answering the fundamental question: what is an architecture, and why is the existing one wrong?

Much of the literature focuses on individual projects' relation to TCP/IP or each other, mostly with focus on a certain paradigm or aspect (e.g. routing, naming, or security). Few question the core assumptions that forms the basis for TCP/IP, for example the layered model, the transport protocols, the established dogmas (e.g. the end-to-end principle), and the APIs.

In this dissertation, we aim to provide an in-depth comparison of a few select IAs. We wish to do so based on fundamental systems and software design principles. Whereas much of the literature, some of which we have reviewed in the previous sections, provide comparisons and/or new IA proposals on a general level, very few dive into the APIs, the core abstractions, and their specific mappings on a detailed level. This work should be a supplement to the existing literature in this respect.

2.2.4.1 Motivation for Selection of IAs for Comparison

In this work, we seek to analyse core IA concepts from a systems architecture perspective. We look for a small selection that allows us to: 1) relate to the current IA, 2) be critical of the

current IA, and 3) represent the FIA research field. Based on these criteria, we have chosen to focus on clean slate approaches, as well as the current TCP/IP. We have selected TCP/IP, NDN, and RINA.

TCP/IP has been chosen because it is the current state of the art, and because any FIA will inevitably be benchmarked against it. CCN/NDN is a radical departure from TCP/IP, and is at the same time a seminal work in the ICN and FIA fields. RINA is highly critical towards TCP/IP, while at the same time it stays in familiar territory because its services and abstractions are similar to TCP/IP. It is influenced by classic research (for example Pouzin 1973 and Day and Zimmermann 1983) that never made it into TCP/IP, providing a good balance towards the two others.

2.3 Principles

In the natural sciences, models describe the real world. If the model does not reflect the real world adequately, the discrepancy will sooner or later become apparent. In such cases, the model is updated, usually after much scrutiny and debate amongst scientists. In this sense, the real world tells scientists whether or not their science is correct.

As Day (2008a) points out, computer science models do not in the same way have a real world to expose their accuracy, because computer science models do not in the same way as the natural science models describe the real world. Rather, computer science models describe an artificial computer world devised by scientists.

While the laws of the real world cannot be changed, the laws of the artificial world can to some extent be changed. Thus, computer science models are in the peculiar situation that they describe a world devised by their own laws.

The question then is: How can we evaluate a system, if the system is defined from the model, and the model depends on the system?

To be so, the model should be based on principles that are sound, in the mathematical sense.

A software system solves some problem. The complexity of the software system is partly due to the problem addressed, and partly due to the particular solution chosen for the problem. Brooks (1987) refers to the former as *essential complexity* and the latter as *accidental complexity*. Essential complexity stems from the *problem space*, i.e., the abstract nature of the particular problem to be solved, which is outside of the control of the system

designer; while accidental complexity stems from the *solution space*, i.e., the characteristics of specific solutions, implementations or decisions which could have been different, which are controlled by the system designer.

When the problem is well-defined and fixed, essential complexity risks being fixed too. Good software design is about minimising accidental complexity. The core principle to do this is *modularity* via *abstractions*.

2.3.1 Modularity via Abstractions

Modularity means decomposing a problem into smaller, simpler, more tractable subproblems, solve each one, and combine the results. In software, this translates into decomposing a large system into smaller bits, called components or modules.

Abstracting is to take a complex reality and remove everything irrelevant for solving the specific problem at hand, thereby building a model without superfluous information that can be used to solve the problem. It can mean that we work in terms of general concepts and ideas close to the problem space instead of in terms of specialised or concretised realisations or instances of those, close to the solution space.

In the object-oriented programming paradigm, abstractions has become concrete: they are known as abstract classes (C#, Java), virtual classes (C++), interfaces (C#, Java), or traits (Scala). They define a class of code and data where some parts of the code or data is left unspecified, to be specified (implemented) somewhere else. These abstractions are still useful because they define the interaction of the class with other classes, even if the details are not available where they are used—i.e., they are extensional.

Effective modularisation can be achieved, if the modules can be devised so that the *coupling* between modules is minimised, and the *cohesion* within each module is maximised (Stevens et al. 1974). Both terms are a measure of the relationships between different parts of a system: Low coupling between modules means that parts of the system that exist in different modules should have weak/few relations, while high cohesion within modules means that parts of the system that exist in the same module should have strong/many relations. The nature of a problem will be a defining factor as to how this can be done.

2.3.1.1 Encapsulation

Encapsulation means to hide detailed information inside modules. D. L. Parnas 1972 showed how encapsulation helps achieve modularity. By encapsulating data and logic that originates

from the solution space, and letting modules only expose abstract interfaces that depend on the problem space, solutions, design decisions, and implementations can freely change inside one module without affecting other modules.

2.3.1.2 Indirection

Indirection means to map identifiers from one namespace to identifiers in another namespace. While identifiers in each namespace might bind statically to external entities (hardware or network locations, variable names, applications names, etc.), the mapping ensures that the external entities linked via the indirection can be relinked dynamically when needed.

2.3.1.3 Extensionality

Extensionality is equivalent to the black box approach in module design. It means, that a module is defined in terms of its interfaces and interactions with external entities, as opposed to internal design or implementation choices. Thus it defines the external equivalence of entities.

2.3.1.4 Generalisation

Generalisation means to formulate general concepts from concrete instances by abstracting common properties. It may involve distilling commonalities from these specific instances. Carl Linnaeus, the Swedish botanist, applied generalisation to create a biological taxonomy with several levels of abstractions to order living things.¹

2.3.2 Black Boxes and Extensionality

A *black box* is an abstract module that is completely defined by the semantics of its interfaces. It is opaque in the sense that nothing about its internals are assumed. Such a black box module is said to be *extensional*.

A module, extensional or not, has two kinds of interfaces: The *required* ones provided by other modules that the module depend on in order to function, and the ones that the module itself *provides* for other modules to use. In schematic figures involving black boxes, the required interfaces are placed at the bottom of the black box, while the provided interfaces are placed at the top. Even though the black box is opaque, the interfaces of the black box are tightly related.

¹And, originally, dead things in the form of minerals.

Several black boxes can be combined into a single black box. That way, the interfaces between the combined black boxes is hidden and becomes part of the invisible internals of the larger black box. The choice as to how and when to consolidate black boxes defines the granularity at which a system is viewed.

The *design* of a system is its black boxes, their interfaces, and their relations. *Architecture* is some times used as synonymous with 'design', however, architecture more correctly denotes a class, type or pattern of design. Both design and architecture depends on the abstraction level, i.e. the granularity of the black box view.

2.3.2.1 Generality

For a class of problems with a shared set of *invariants* and a set of *variables* which may vary for each problem, a solution

Software depends on *invariants* and *variables*. The former is what is assumed not to change, while the latter is what is assumed to change. Generality of software is a measure of the proportion between the two: The more invariants a software module relies on and the less variables it can handle, the more specialised or less generalised it is, and vice versa.

Generality of software is a trade-off: A more general solution is typically more complex than a more specialised one, because it takes more work to take into account a variable range of possible values than a single, fixed one. On the other hand, a more general solution solves a wider range of problems.

Generality is connected with robustness, which is how well software account for variable situations.

2.4 Protocols and Interfaces

In this section, we attempt to answer the following questions:

1. What is a protocol?
2. What is an interface?
3. What are the differences?
4. What are the commonalities?

The answers we provide are based on Day 2008a, Chapter 2, as well as on subsequent conversations with John Day. We use finite state machines (FSMs) as a central building block, although other computational models could have been used as well.

An FSM is a model of computation consisting of a finite set of states and rules for transitioning between these states. Day 2008a describes FSMs as defined by:

- An input alphabet: $A = \{a_1, \dots, a_m\}$
- A set of states: $S = \{S_1, \dots, S_n\}$
- An output alphabet: $O = \{o_1, \dots, o_p\}$
- A transition function: $F_1 : A \times S \rightarrow S$
- An output function: $F_2 : A \times S \rightarrow O$

The number of states is, in the worst case, in the order of magnitude of the number of possible inputs, which is the number of accepted permutations of the input alphabet. This number can be extremely large, so, in practice, a simplified version of FSMs is used in which all possible input-dependent states are reduced to form a smaller set of major states in which minor variations that depend on the input are contained. From here on forward, when we refer to FSMs, it is this simplified version we mean.

2.4.1 Protocols

We here briefly summarise the concept of a protocol as defined in Day 2008a, Chapter 2: Two systems need to have a common conceptual schema in order to be able to communicate. The shared schema consists of the concepts of the world and the things in it the two systems agree they want to talk about. This schema is embodied in an FSM in each system. The set of rules and procedures that each FSM is required to follow to maintain the coordination of their shared schema is called a *protocol*. It includes the shared communication syntax and semantics. Sometimes we call an FSM that embodies a protocol for a PM to distinguish it from a general FSM, which may or may not embody a protocol.

When an entity (e.g., a system, FSM, or PM) communicates with another, we adopt the terminology from Day 2008a and say that the other entity is the *apposite* entity.

The communication does not exist for the sake of the PMs: Something *drives* the PM in each system. Generally, this is another FSM, existing in the same system. That FSM may be driven by other FSMs and so forth. While the PMs of two separate systems communicate via the exchange of small data messages, the driving FSM and the PM of each side communicate via an *interface*, which we describe in the next section.

So far, we can picture the communication between two systems as shown in Figure 2.3.

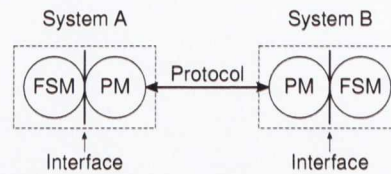


Figure 2.3: Two systems communicating via a protocol.

2.4.2 Interfaces

While protocols facilitate communication between FSMs in different systems, interfaces facilitate communication between FSMs in the same system.

The simplest interface between apposite FSMs is for them to have a shared memory region which they can read from and write to. They need to be very tightly bound in order to coordinate when to read from and write to the shared region. This coordination can be done by each FSM voluntarily handing over control to its apposite when needed—i.e. each FSM *blocks* while the other executes: They run synchronously and never in parallel. Alternatively, they might run asynchronously and in parallel, and use hardware-enabled synchronisation primitives for coordination instead. In general, distributed systems run in an asynchronous context. For both cases to work, both FSMs expect and depend on very specific behaviour from its apposite to a degree where one will not function without the other. While we keep the two conceptually separated, the two FSMs do in a sense combine into one abstract FSM.

2.4.3 Protocol, Services and Extensionality

Two PMs interacting via a protocol can be seen as two black boxes with an interface between them. Each PM require the protocol interface provided by the other. In addition, they each expose an API for clients of the protocol to use. The protocol interfaces and the client interfaces are tightly related; the purpose of the client interface is to control the PM to drive the protocol.

This view can prompt system designers, e.g., Jacobson et al. 2009, to believe that the protocol should be the primary invariant, and that clients are users of the protocol and need to adapt to it.

However, we can instead view the clients as users of some service that enables them to communicate. We can insert an ‘adapter’ black box that abstracts the protocol-specific PM API as a service API. Instead of keeping the protocol invariant, we can keep the service API

invariant, and allow the protocol to vary. This reverses the roles: The protocol now exist to facilitate the service API, not the other way around. The clients become dependent on the abstract service rather than the protocols.

This change of perception provides two benefits: The service as a whole can, from the point of view of the clients, be viewed as a black box, encapsulating the protocol, the PMs, and the adapters. The clients (or the developers of the client software) do not need to be concerned with protocol details. Secondly, the service API directly specifies what level of service the protocol should provide.

When an application interacts with a network protocol via an API, we have FSMs at three levels: i) The network PM, ii) A black-box FSM, and iii) The application-side API FSM. The network PM by definition drives the network protocol. The black box FSM drives the PM and it also provides the application-facing API. Thus, it works as a black box between the PM and the application-side API FSM, bridging the two. The application-side API FSM drives the black box FSM. This is depicted in Figure 2.4. Abstractions cannot be introduced

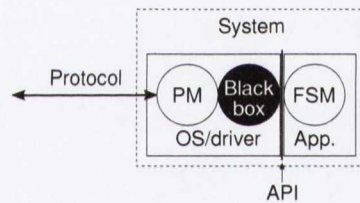


Figure 2.4: The Network PM

between FSMs because two interacting FSMs need to be tightly synchronised with respect to both timing and schema. Therefore, abstractions need to be designed into an FSM as the difference between the schemas of the interfaces of that FSM. The black box FSM does exactly that: It abstracts the low-level interface to the PM in a high-level API to the application. The black box FSM might consist of several FSMs. We do not know that; all we care about are the interfaces and the externally visible behaviour of the black box.

2.4.4 Summary

Both interfaces and protocols serve to enable communication between application components. Whereas interfaces facilitate communication between components in the same system, protocols facilitates communication between components across systems. The interaction via protocol can be made to look like the interaction via an interface by abstracting it away

by means of one or more intermediary black-box FSMs as described above. In that aspect it does not matter to applications whether or not they communicate directly via an interface or via a protocol.

However, in practice it does matter: The applications communicating via a protocol over a network will have to be prepared to handle both latency and loss of communication. These are generally not issues applications communicating directly via an interface need to consider—if an interface fails to communicate or latency becomes problematic, something fatal such as memory corruption or system overload is likely at play, and application failure is accepted. If, however, network communication is involved, these situations will arise more frequently and will not, in general, be caused by fatal system errors, and the application should be able to handle such situations.

2.5 Location

In the real world, location is something physical: A cup is placed on a table, in a house, on a street, in a city, in a country. Implicitly, on planet Earth, in our solar system, in the Milky Way. In other words, the location of something reveals which entity one must reach first in order to reach that something that is one's destination. In the physical world, the entity one must reach first (for example, a country) can be of greater scope than the entity that is located inside it (for example, a city). Greater scope means that there is a many-to-one relationship: Several cities can be in one country, but a city can only belong to one country. The many-to-one generalisation have notable exceptions, such as border-cities (for example Niagara Falls) where a city is located in more than one country, or city-states (for example, Singapore), where the city is not located in a country at all, or alternatively that the city and its country are the same.

In computer science, location is used in a similar way, but the entities are different. Some entities belong to the computer science realm and are not physical, for example domain, process, program, or directory, while others are physical, but have computer science connotations, for example data center, host, disk, or memory. Some are intermixed and vague, for example 'the cloud' or 'cyberspace'.

Let us from these comments note the invariant properties of the term 'location':

1. The location of some entity is another entity relative to which the entity can be reached.
2. *Transitivity*—A location of a location of an entity is also that entity's location, just like a

house in a city in a country can be said to be located in that country.

3. *Irreflexivity*—A location cannot be its own location.

The latter seems to contradict the example of Singapore: After all, Singapore (the city) is located in Singapore (the country). However, in reality, there is no contradiction: Either Singapore the city and Singapore the country are in fact considered different entities (where Singapore the city is located in Singapore the country, but not vice versa), or they are conflated into Singapore the city-state (and it makes little sense to say that Singapore the city-state is located in Singapore the city-state).

This means that locations form hierarchies, or more precisely DAGs. It also means that a location of an entity does not make a entity reachable on its own: You have to know how to reach the location too.

2.6 Properties

We choose to evaluate network architectures based on five properties:

Mobility The extent to which the service is disturbed when entities change location, and how easy it is to recover from such disturbances.

Flexibility How easy it is to adapt the service, for example changing protocols, routing, naming, or security aspects.

Security How secure the service is, e.g. how well it can be protected from foes outside and inside the system.

Scalability How well the service scales and under which circumstances.

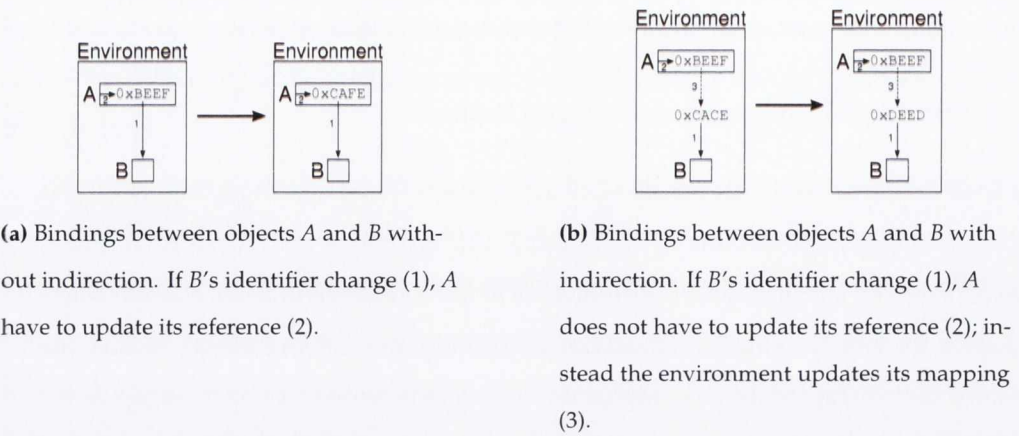
Simplicity How simple the design is, e.g. in terms of ease of implementation and understanding, and of intra-module cohesiveness and inter-module coupling.

2.6.1 Mobility

When object *A* refers to object *B* by its identifier `0xBEEF`, there are two bindings in place:

1) The object *B* is bound to its identifier `0xBEEF` in some environment common to *A* and *B*, and 2) The identifier `0xBEEF` is contained in object *A*'s scope. This is shown in Figure 2.5a.

If *B*'s identifier contains information about how to reach *B* (in which case, we might call it an *address*), or if it is a hash digest or other kind of derivative of the data and state of *B*, binding 1 is predetermined and cannot be chosen freely; and it might change.



Therefore, in order to change the mapping from *A* to *B*, binding 2 needs to be updated, which requires changing object *A*. If there are multiple referrer objects like *A* containing the identifier of *B*, they all need to change. This does not scale for a large number of referrer objects.

For example, if `0xBEEF` is the physical memory address of *B* and it is used in *N* referrer objects, then *B* cannot change location in memory without the address changing, which requires that the *N* referrer objects updates their saved identifier also.

The solution used in OS'es is *virtual memory*. It provides an *indirection*, i.e., a flexible binding, in the path between *A* and *B*. This is shown in Figure 2.5b as binding 3. When binding 1 changes, instead of updating binding 2 in all referrer objects, we update the one binding 3 in the environment. This makes the update scalable, manageable, and transparent to the referrer objects.

According to J. Saltzer 1993, indirections are also the solution in networking: Names, identifying 'what you want', redirects to addresses, identifying 'where it is', which redirects to routes, specifying 'a way to get there'. Day 2016 refines this further, pointing out the important difference between addressing nodes and PoAs, which J. Saltzer (1993) does not. Addressing nodes instead of PoAs reduces routing table sizes, allows late-binding of paths, and ultimately enables multihoming.

2.6.2 Flexibility

Flexibility is the ability for system to adapt to changing needs. This can, for example, be achieved by pluggable policies or by enabling configuration of semantics, syntax and behaviour of certain aspects of the system. In the IA field, the term 'evolvability' is sometimes used, for example in SDIA (Raghavan, Koponen, Ghodsi, Casado, et al. 2012) and in XIA

(Anand et al. 2011).

Flexibility is always a trade-off with the constraints imposed by an IA. The constraints define the semantics of an IA and enables users of the IA to expect and rely on those semantics. If an IA imposes too many constraints that are unchangeable, it may become rigid and inflexible. On the other hands, if there are too few constraints, semantics become indefinable or too imprecise. Choosing the right constraints and the level to which they can be configured is paramount to making an IA useful.

As an example, let us discuss flexibility with respect to addresses. The semantics of addresses, i.e., what they refer to (for example, hosts or PoAs, as we discuss in Section 7.4), is a decision and constraint that is important for the system and for clients to rely on, because it defines core aspects of the behaviour of the system and how clients need to relate to it. The syntax of addresses, on the other hand, is not important to clients. It may influence routing state updates, if the syntax contains topological information (as IPs do, for example) so routers may depend on the syntax, but from a client's perspective, it is just a reference to a peer and the actual syntax is not important. The fact that it refers to a peer (its semantics) is important.

2.6.3 Security

We base our discussion of information security in network architectures on five security pillars:

Confidentiality Prevention of unauthorised access to information.

Integrity Detection of unauthorised or accidental change to information.

Availability Ensuring that information is available when and where needed.

Provenance Verification of origin/author of information.

Authentication Verification of the identity of entities.

These are interdependent. For example, confidentiality does not make sense without authentication, and authentication requires integrity to detect identification spoofing.

In data networks, there are standard solutions to many of these pillars. For example, confidentiality of information in transit is typically achieved using symmetric-key cryptography in combination with an authentication and key-exchange method based on asymmetric-key cryptography.

The integrity of information in transit is achieved using check-sums or hash-functions, while detection of unauthorised changes relies on provenance, typically achieved using

digital signatures, which also relies on cryptographic methods.

A network architecture can make it more difficult or not to achieve these security properties. For example, if there is no way to assess the provenance of a PDU, authentication is hard or impossible; and if there is no way to prevent unauthorised access to resources, availability may suffer.

We discuss the security aspects of each network architecture as we go along.

2.6.4 Scalability

Particularly for distributed systems scalability is important. Scalability is the ability of the system to perform when the system grows in size. For IA, the size can be defined as a function of routers, connected devices and/or hosts.

If a system is not designed with scalability in mind from the beginning, its performance and/or correctness may suffer above a certain size threshold.

2.6.5 Simplicity

Simple systems require fewer steps to implement, understand, and reason about. Simplicity is difficult to qualify, but in terms of a system consisting of interdependent modules, it may be defined as a function of the number of types of modules, the number of module instances, the interdependencies between modules, and the flow of data between modules.

Chapter 3

Taxonomy

Based on Chapter 2, we here define the taxonomy with which we categorize IAs. The taxonomy consists of a set of principles and a set of properties. An IA is placed in this taxonomy based on to what extent the IA adheres to the principles and to what extent it satisfies the properties.

3.1 Principles

We choose to base our taxonomy on the following four principles:

1. Indirection (Ind)
2. Encapsulation (Enc)
3. Generality (Gen)
4. Extensionality (Ext)

The abbreviation in parenthesis is used in some tables.

Indirection is used to translate implementation-specific concepts to general, model-specific concepts, or to translate concepts from one component to concepts from another component. It is also used to map identifiers between different scopes. This mainly helps to keep the coupling between components low.

Encapsulation creates high cohesion within components (D. L. Parnas 1972; D L Parnas et al. 1985; Stevens et al. 1974) and is used to shield internal component data and semantics from other components. Using encapsulation usually involves indirections as well, whereas using indirections does not necessarily means that concepts are encapsulated. We also include scoping in the term.

Generalisation means to create modules with many variables rather than ones with many

fixed choices. This reduces the number of invariants, and makes a module more adaptable. Generalisation may involve both encapsulations and indirections to hide implementation-specific choices.

Extensionality is when the design as a whole is specified in terms of abstract concepts rather than implementation-specific ones. It is centred around the black-box concept, where a design is built with general, abstract components and interfaces.

These four principles are not mutually exclusive. Extensionality typically involves all the others; generality is born from the extensional approach, and is typically realised using encapsulation and indirection; and encapsulation of data that at the same time needs to be shared in some form requires indirections.

3.1.1 Justification

These four principles form the basis of our IA evaluations. Because they are to be used to evaluate multiple and different IAs, they should be independent of any single IA, and in particular, independent of the current IA. This is contrary to much existing literature, which defines principles that depend on a concrete solution, that are goals or objectives, or that rely on the principle of the philosophy of TCP/IP, which is the solution that is sought to be replaced.

We have chosen these principles because they are based on the core concepts of *modularity* and *abstractions* as shown in Figure 3.1.

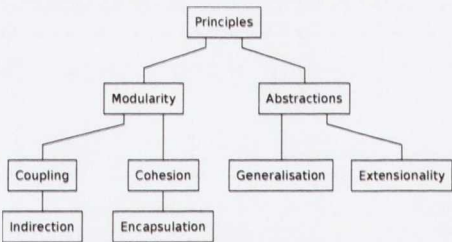


Figure 3.1: Our view of the main relations between chosen taxonomy principles and the core principles of modularity and abstractions.

Modularity and abstractions are the core concepts of the seminal works of D. L. Parnas (1972), D L Parnas et al. (1985), and Stevens et al. (1974) on software architecture.

Because IAs are software architectures and the only difference between a generic software architecture and an IA is the purpose of the system being architected, and because the purpose of a system should not influence the principles behind its architecture—they should

transcend the purpose—we base our principles on these seminal works. Thus, our principles are applicable to software architectures in general, and therefore also to IAs in particular.

We do not believe we can choose more specific principles without introducing biases stemming from assumptions as to *how* the architected solution fulfils its purpose. In particular, we do not believe we should choose principles that depends on the specifics in TCP/IP or on any other IA, for that matter.

If we are to be able to evaluate all possible IAs, we cannot assume anything as to how the systems are construed, and we need a set of principles reflecting this. Hence our principles are generic for all software architectures and should therefore also apply equally to all IAs.

3.1.1.1 Architectural Principles in TCP/IP

The current TCP/IP IA has evolved over many years and its architecture and principles are more descriptive than normative, meaning that the principles describe what has been successfully done in the past on the assumption that this should also guide the future architecture (Carpenter 1996). In fact, architectural principles are not considered important compared to running code.¹

Many principles of Carpenter (1996) relate to the TCP/IP IA. For example, the end-to-end principle (Carpenter 1996, Section 2) puts data connection state at the endpoints rather than in the middle, while routing state may reside in the network, because this partitioning of state has been successful for TCP/IP. It assumes that state can be partitioned like this, which is not generally the case—in NDN, for example, the routing state used for data packets is transient and depends on the interests that have travelled the network; in active networks (Tennenhouse et al. 1997) and in SDN (Haleplidis et al. 2015), the routing state changes depending on the data flow.

Some principles describe desirable properties of the end product, rather than architectural principles to adhere to during design, for example scalability or performance (Carpenter 1996, Sections 3.3–3.4). Some are guidelines regarding the standardisation or design process, for example that “nothing gets standardised until there are multiple instances of running code” (Carpenter 1996, Section 3.14) or that unpatented technology should be preferred (Carpenter 1996, Section 5.1). Some are design patterns or solution guidelines, for example the section regarding names (Carpenter 1996, Section 4), which mandates certain formats for names and addresses (Carpenter 1996, Section 4.4). However, some principles are general

¹[The Internet’s] evolution depends on rough consensus about technical proposals, and on running code. Engineering feed-back from real implementations is more important than any architectural principles. From Carpenter (1996, Section 2.4)

and address how things should be architected, for example the principle of modularity (Carpenter 1996, Section 3.6).

Bush and D. Meyer (2002) expands on Carpenter (1996), and includes additional principles, for example regarding coupling (Bush and D. Meyer 2002, Section 2.2.2), layers (Bush and D. Meyer 2002, Section 3), and packet versus circuit switching (Bush and D. Meyer 2002, Section 5).

Clark (1988) defines the primary goals of the Internet. It does not directly define principles which an architecture of a system achieving those goals should adhere to.

However, many of the concepts and philosophies rely on underlying architectural principles. For example, the concept of layers and the end-to-end philosophy is based on extensionality and modularity.

3.1.1.2 Architectural Principles in Other Related Work

CCN/NDN Jacobson et al. (2009) do not directly define the architectural principles they adhere to. They do mention decoupling in their abstract as something desirable, but otherwise they list goals to achieve: scalability, security, and performance. They assume several patterns of TCP/IP: for example, they assume that the layers, the autonomous systems (AS'es), and the routing paradigm in the TCP/IP-model subsist, and that scalability should be achieved via name aggregation.

Zhang, Afanasyev, et al. (2014) do not define architectural principles either, but argue that the problem to be solved is that of a distribution network rather than a communication network, and state that the former is more general than the latter. This is a guiding philosophy throughout the NDN literature, however, we would characterise it as a self-imposed constraint rather than a principle.

RINA Day (2008a) discusses principles of science and the philosophy of science, referring to, for example, Newton's *Principia* (Newton 2014) and Wittgenstein's *Tractatus* (Wittenstein 2012). He discusses patterns of network architectures, but like other related work, he does not directly define a set of architectural principles, other than the general principles of science.

He does define a series of patterns suitable for network architectures. These patterns can be traced to basic principles. For example, the philosophy of separating mechanism from policy (Day 2008a, Page 39) is an application of encapsulation (of the policies) and generalisation (of the mechanisms).

3.1.1.3 Commonalities

The commonalities that the literature reviewed in previous sections share, are those of ‘sound’ software architectural principles revolving around modularity and abstractions. Many of the goals, patterns, and principles directly or indirectly reflects this, as we have provided some examples of above.

This fact, along with our discussion at the beginning of Section 3.1.1 is our justification for choosing the four principles, ultimately derived from the seminal works of D. L. Parnas (1972), D L Parnas et al. (1985), and Stevens et al. (1974).

3.2 Properties

We choose to base our taxonomy on the following properties:

1. Flexibility (Flx)
2. Mobility (Mob)
3. Scalability (Scl)
4. Security (Sec)
5. Simplicity (Sim)

Flexibility is the ability to change, either when the problem-space is changed, or when collaborating components or subcomponents change. In Raghavan, Koponen, Ghodsi, Casado, et al. 2012 and Anand et al. 2011, it is called evolvability. The opposite of flexibility is rigidity.

Mobility is the ability for some components to change location. TCP/IP’s lack of support for mobility has led to several augmentations to the current IA, e.g. MIP (Perkins 2010) and LISP (Farinacci et al. 2013), and there is general agreement that mobility support is an indispensable property of a IA.

Scalability is the ability to grow the number of users or nodes without the system becoming inefficient or collapsing (Tanenbaum and Wetherall 2011). Scalability is essential for any large distributed system. TCP/IP, for example, struggles with scalability in its addressing design, which has lead to the development of IPv6 (Deering and Hinden 1998).

Security is the ability to facilitate or support the five security pillars: Confidentiality, Integrity, Availability, Provenance, and Authentication. Many IAs puts special emphasis

on one or more security aspects, for example, ICNs (including NDN (Zhang, Afanasyev, et al. 2014), DONA (Koponen, Chawla, et al. 2007) and NetInf (Dannewitz et al. 2013)) are focused on provenance.

Simplicity has two aspects: One is the simplicity of the IA itself, which affects how easy it is to change, understand, and implement; the other is how simple it is to use, i.e. how simple its user applications can be. An IA that is inherently simple, but increases the complexity of its user applications, might still lead to an overall complex system.

See also Bush and D. Meyer 2002.

Similar to the list of principles, the list of properties is not an objective choice. We base our choice on what the FIA literature has focused on. We argue that even though it is not exhaustive, and other or more properties could have been included, there is consensus that these are important and central properties.

3.3 Usage

We use the taxonomy to assess and score individual architectures according to how well they employ each principle and to what extent they achieve or possess each property. We score the architectures from -2 for the worst score, meaning that an architecture does not, or only very poorly, employ a principle or possess a property, to 2 for the best or highest score, meaning that an architecture consequently or mostly employs a principle or possess a property.

3.4 Summary

We have defined the taxonomy with which we assess IAs in the remainder of this dissertation. The taxonomy consists of four principles: 1) Indirection, 2) Encapsulation, 3) Generality, and 4) Extensionality; and five properties: 1) Flexibility, 2) Mobility, 3) Scalability, 4) Security, and 5) Simplicity.

Each IA we assess is scored in each of the principles and each of the properties with a score ranging from -2 (worst) to 2 (best).

3.5 Review and Justification of Evaluation Approach

In this work, we have chosen to evaluate the IAs we assess based on the four principles and five properties laid out in earlier in this chapter. The principles revolves around modularity, and are chosen to be independent of network and software architecture, but are principles that are relevant in the design of any complex system. Nevertheless, they are used extensively in the computer software literature, for example in the object-oriented design paradigm, where these principles shape languages and patterns to foster modularity in software systems.

This evaluation approach may seem unobjective. After all, we could have chosen a different set of principles, a different set of properties, or we could have used a set of concrete metrics instead. We assert that *no* evaluation criteria are absolute—they will always be relative to a set of standards, which in the end have to be justified. Our justification is that our chosen principles are relevant for all system designs and constitute “good practices” for designing sound systems, and that our chosen set of properties are relevant to IAs in particular and are desirable properties of an IA.

As such, the assertion we make in this work is the following: if an IA adhere to our chosen set of principles, which we deem “good practice”, it will, to a high degree, inhabit the chosen properties, which we deem desirable.

Chapter 4

The Classic Internet

The classic Internet architecture encompasses the IP version 4 (IPv4)-based TCP/IP architecture including DNS. We explain the TCP/IP model, the APIs, and following that we examine how and where the four principles of our taxonomy are employed. We find that the principles are poorly employed and that, as a result, the classic TCP/IP architecture lacks many desired properties.

4.1 Model

When it comes to scoping, there is only local (link) scope, or global scope in TCP/IP. Therefore, scope is artificially introduced via subnets and AS'es, with different routing mechanisms and algorithms working within each scope—but all within same layer. Network address translation (NAT) is an example of how scoping is desirable for isolation and security purposes.

4.1.1 Layers

The Internet model used in many textbooks (for example in Kurose and Ross 2013; Tanenbaum and Wetherall 2011) has five layers numbered from 1 to 5, shown in Table 4.1. We also sometimes refer to this model as the 'TCP/IP model' or simply as 'TCP/IP'. This model is descriptive rather than normative, meaning that it has been conceived as an after-thought to the Internet with its specific protocols, rather than as a general blueprint for other networks with possibly different protocols. The consequence is that the model is closely tied to the protocols used, and a layer number or name and its protocol are used interchangeably. Also, some protocols (such as MPLS, is referred to as 'a layer 2.5 protocol') do not fit into the model.

A typical view of the model is shown in Figure 4.1. The figure shows the components

Layer	Name	Scope	Mechanisms	Typical protocols
L1	Physical	Physical medium	Signal, modulation, interference handling	Manchester Encoding
L2	Link	Small network or link	Error handling, relaying and primitive routing, link retransmission (for some unreliable mediums)	IEEE 802.3 (Ethernet), IEEE 802.11 (WiFi)
L3	Network, Internet	Networks of various sizes, from home LANs to the global Internet	Error handling, fragmentation, large-scale relaying and routing	IPv4, IPv6
L4	Transport	Similar to L3's	Error and flow, fragmentation, reliability, retransmission, ordering	UDP, TCP
L5	Application	Similar to L3's	Application-specific	HTTP, FTP, IMAP, DNS

Table 4.1: Layers in the Internet model.

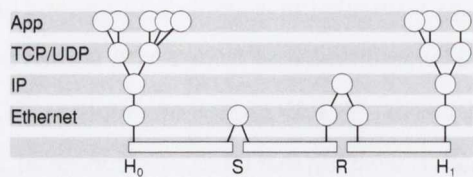


Figure 4.1: TCP/IP model.

Layer	Component	Identifier	Uniqueness	Scope/relative to
L2	Ethernet PM	MAC address	Global	Other Ethernet PMs in LAN
L3	IP relay	Ethertype	Local, well-known	Ethernet PMs in same host
L3	IP PM	IP address	Global	IP relay
L4	TCP (or UDP) de-mux	Protocol number	Local, well-known	IP instance
L4	TCP (or UDP) PM aka. socket	Port number	Local, well-known or random	TCP (or UDP) de-mux

Table 4.2: TCP/IP identifiers, their scope, and their users.

(circles) and the various layers (light horizontal boxes). Several components are gathered at each system or host.

Here, two hosts, H_0 and H_1 are connected via IP in layer 3 (L3). The IP link is relayed via the L3/IP router R , which connects two Ethernet LANs: one spanning R and H_1 , and the other spanning H_0 , S , and R . The second LAN spans two physical media, connected via the layer 2 (L2) Ethernet switch S . On top of L3, the layer 4 (L4) transport layer provides end-to-end communication service.

4.1.2 Naming and Addressing

A model of an example host’s stack has following components:

- 1. PoA
- 2. Ethernet instance
- 3. IP relay
- 4. IP instance
- 5. TCP relay
- 6. TCP instance (port/socket)
- 7. Applications

They are identified as shown in Table 4.2.

The Ethernet PM is bound to a network medium via a network interface card (NIC). We call the part of the medium where the NIC is attached for the PoA.

The Ethernet PM is identified via its MAC address, which is an identifier with which other Ethernet PMs on the same LAN can reach it. One IP PM is bound to each Ethernet PM (except in special bridge configurations). There is a one-to-one relation between IP addresses, MAC addresses, NICs and PoAs; and we say that the IP address identifies the PoA (despite the common misunderstanding that it identifies the host).

The applications and the API between applications and the stack exist outside of the TCP/IP model. TCP/IP delivers PDUs to a socket and leave the rest to the OS and application. TCP/IP PDUs identifies a socket via a relative path name anchored in the IP layer specifying the path from the IP layer to the socket. The path name is a three-tuple that consists of an IP address (to identify which IP PM the socket belongs to), a protocol number (to identify which transport demux type the socket belongs to), and a port number to identify the socket. The IP address and protocol number is part of the IP protocol control information (PCI), while the port number is part of the transport (e.g. TCP or UDP) PCI. However, TCP/IP uses *pseudo-headers* that include PCI fields from both IP and TCP/UDP necessary for transport protocol processing.

Once a PDU has reached the IP layer of the host where the destination socket is located, the path-name contained in the pseudo-header uniquely identifies the socket. Thus, the question of routing a PDU to the IP layer of that stack remains.

The IP address reveals whether the destination of the PDU is in the current network or not. If it is, the IP address contained in the IP PCI is mapped to the MAC address of the Ethernet PM where the IP PM is located. Such mappings are stored in all hosts and IP routers, and kept up to date with Address Resolution Protocol (ARP). If the destination is not in the current network, the IP will map to the MAC address of a gateway connected to another LAN that is closer to the destination, and the process is repeated. Routing algorithms keep routers and gateways updated with the IP addresses of other routers and gateways so that their MAC addresses can be looked up for PDU forwarding.

4.1.3 The Socket API

In 1983, Berkeley Software Distribution (BSD) 4.2 included an implementation of TCP/IP along with a general IPC API built around the concept of *sockets*. The socket API quickly became the *de facto* TCP/IP API, and has later been standardised as the POSIX socket API (*POSIX.1-2008, 2013 version* 2013).

A socket is a component that contains the state for a PM. For connection-oriented protocols such as TCP or Unix local IPC, the peer socket is statically bound; for other protocols it is not.

The socket is managed by the kernel and referred to by the application via a file descriptor. It is controlled by the application via the socket API. Because the socket is basically a PM, which varies dependent on the protocol, the API differs depending on the protocol. A TCP socket and a UDP socket, for example, have different semantics. The consequence is that

applications are tailored to use a specific API and a specific protocol.

4.1.3.1 Operations and States

While a connection itself might be symmetric (depending on the protocol), connection establishment is always asymmetric: It is always the case that one side requests the connection establishment and the other side responds to it. We typically refer to the requester as the client and the responder as the server.

In the TCP socket API, a new connection to another socket with a known endpoint identifier (EID) is requested by the client first invoking `socket` followed by `connect`. The new socket is automatically bound by `connect` to a local EID chosen by the OS, but the client usually need not care about the assigned EID because it interacts with the socket using its file descriptor. This is lightweight and fairly logical.

Responding to an incoming connection request is more elaborate. The server invokes the `socket` operation to allocate a new socket. It then binds it to an EID with `bind`. The EID needs to be known by the client because the client must use the same EID in its `connect` operation. The `listen` operation puts the socket in a listening state so that incoming connection requests are queued. At this point, clients may connect to the server using the `connect` operation on their side. The `accept` operation on the server spawns a new socket for a waiting connection request. The connection is established, and the new socket is bound to an EID assigned by the OS.

Why is this rather complicated set of semantics necessary? Why not simply `socket` followed by `accept`, to mirror the client API?

The answer is two-fold. First, `bind` is necessary because the server actively chooses which EID it should be accessible at. `listen` is always preceded by a `bind`, so it could have been incorporated into the `listen` operation given a relevant `eid`, or, alternatively, the binding could have been a part of the `socket` operation, providing for both OS and application assigned EIDs. Neither was done.

Secondly, there can be exactly one established connection per socket, so in order to establish multiple connections to multiple clients, one socket per connection needs to be created, each with their own unique EID. The listening socket cannot itself be used for the new connections because it can only be bound to one EID, and this should be the EID known to clients. Therefore, `accept` is necessary to spawn one new socket with an OS-assigned EID per incoming connection request.

4.1.4 Enrolment

In the Internet architecture, enrolment in L3 of a new PoA entails that it has the following information: Its own IP address, the address space of the local network (in the form of a subnet mask), and the address of the first-hop router connecting the local network to other networks. The addresses of one or more DNS servers for domain name resolution can be argued to be a part of the enrolment information as well, because without it the many applications dependent on DNS would not function; however, strictly speaking, it is not part of TCP/IP.

The enrolment phase in the Internet architecture is equivalent to configuring the PoA with this information. TCP/IP was designed without an automatic solution, so this had to happen manually until automatic mechanisms such as Bootstrap Protocol (BOOTP) (Croft and Gilmore 1985) and later the Dynamic Host Configuration Protocol (DHCP) (Droms 1993) was devised.

There is no authentication built into the Internet architecture's enrolment phase. This means that authentication either happens before enrolment; or that it happens after enrolment, but access is restricted until it has happened.

802.1X (*IEEE 802.1X* 2010) is an example of the first case: It handles authentication of the L2 link that is required for L3 enrolment, which happens after authentication. This, or other methods such as WiFi Protected Access II (WPA2) (*IEEE 802.11* 2007) that work at or below L2, are used for IEEE 802.11 (WiFi) authentication.

The second case authenticates using higher-layer protocols, for example PANA (Forsberg et al. 2008) or web-based authentication. The use of higher-layer protocols for authentication requires enrolment in L3 to have happened already, so enrolment initially gives only limited network access, which is then expanded to the appropriate access upon authentication. This involves using cross-layer firewalls or other network management mechanisms that can dynamically affect forwarding at some crucial intermediary network access point.

4.2 Principles

4.2.1 Indirections

In TCP/IP, for two applications to communicate, each attaches to a socket, and the communication then happens between the two sockets. The flow between the applications is equivalent to the flow between the sockets, and the flow endpoint identifier (FEID) of the flow are the socket identifiers, which consist of a three-tuple with IP address, transport

protocol number, and port number. That is the immediate abstraction from the point of view of the application and is shown in Figure 4.2.

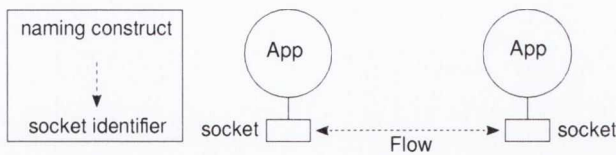


Figure 4.2: Flow between applications is equivalent to the flow between sockets. The FEIDs of the flow are the socket identifiers.

There is an optional indirection from naming constructs in the application, which can be for example URLs, to socket identifiers. This indirection is managed by applications and typically partly relies on DNS, which translates fully qualified domain name (FQDN) to IP addresses. The non-IP part of the socket identifier is typically deduced directly from the URL or from a pre-assumed transport protocol and port number.

That the indirection is managed by applications means that the application decides when and how to invoke the translation, typically via URL parsing in combination with the `getaddrinfo` POSIX API (Gilligan et al. 1999). When a socket identifier changes, which can be caused by an IP change, existing flows become invalid and the mapping in DNS needs to be updated. Applications may eventually find out the new socket identifier from DNS once changes has been propagated through DNS. How long this takes, depends on DNS record settings and on when the application queries DNS, and can be anywhere from a few minutes to several hours.

From the network point of view, the sockets are the connection endpoints. Thus sockets double as flow endpoints and connection endpoints, and socket identifiers are used as both FEIDs from the point of view of the applications and as connection endpoint identifiers (CEIDs) from the point of view of the protocols. This means that an application flow is intrinsically bound to a single network connection during its lifetime: There is no indirection.

The socket identifier and the IP address are intrinsically bound given that the latter is part of the former. In the network, there is an indirection from IP addresses to network PoAs. The IP-PoA indirection is distributed in IP routing tables as well as in the network configuration of hosts and applications.

When an application or a host moves in the network to a new PoA, either the IP of the application or host can be left unchanged, which requires that routing tables are updated with the new IP-PoA mapping, or the IP of the application/host can be updated, which

require configuration updates and invalidates existing socket identifiers and thereby existing flows/connections. The former requires routing table updates across potentially large areas of the network and might violate aggregation, and might therefore not be feasible. The latter is therefore typically preferred. This requires the mobile application or host to be configured with a new IP address, it requires DNS to be updated with the new IP address, and it requires other applications to lookup the new IP in DNS to deduce new valid socket identifiers. Also, existing flows/connections breaks, and will need to be reinitiated by the applications.

4.3 Evaluation

Here we list some of TCP/IP's fundamental design choices and the effect they have. We summarise the resulting issues in Section 4.3.6.

4.3.1 Connection Management

APs manage connections directly via their PMs. A PM is manifested by means of a construct called a 'socket', which is exposed to APs via the socket API.

4.3.2 Endpoint Identification and Binding

The socket is the endpoint. The EID used in TCP, UDP and many other transport protocols is a tuple consisting of an IP to identify the network PoA, and a protocol and port number, to identify the specific socket. This forms a path-name specifying the path through the stack leading to the socket. The EIDs are also used as socket identifiers in the socket API.

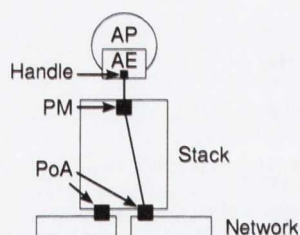


Figure 4.3: A TCP/IP stack and application. Applications bind statically to the socket/PM, which binds statically to a PoA. If either the PM or PoA becomes invalidated, the application needs to recreate the bindings.

4.3.3 Application Names and Addresses

We here define human-friendly names as names that contain a description of the things they name.

In TCP/IP, there are no human-friendly names. Instead, DNS has been devised as a service that (primarily) resolves human-friendly domain names to the IP-addresses needed to build the EID tuples used in TCP/IP. The applications perform this name resolution in a separate step before using the socket API to establish connections. The EIDs used as inputs to the socket API have a structure that reflects the structure of the network stack, and the IP addresses used in the EIDs have a structure that reflects how IP does routing.

4.3.4 Layers and Protocols

In TCP/IP, there is in principle a fixed number and ordering of layers. Layers are distinguished by their functions, and they are facilitated by different protocols. L2, L3 and L4 each expose one or (in the case of L4) several different socket API to applications, but with the kernel-space/user-space boundary in the POSIX API specification they are not meant to be used effectively as inter-layer APIs.

4.3.5 Routing and Renumbering

Routing is done on IP addresses, which are assigned to PoAs rather than hosts. Also, IP addresses are exposed and used by upper layers, including the transport layer (via the EIDs in the protocols) and applications (via the EIDs in the API).

4.3.6 Summary

While the individual design choices in TCP/IP might seem innocent enough, in combination they create serious issues with:

- Mobility
- Scalability
- Security
- Flexibility

4.3.6.1 Mobility Issues

The mobility issues are caused by the combined choice of binding the EID to the PoA via the IP, using the EID to identify sockets, and letting the application manage socket failures. The result is that sockets and thereby connections fail when PoAs change, and that failure handling is up to the application, which means that there are no guarantees for a successful recovery—especially due to DNS being very slow to propagate updates to entries.

One solution is to keep the IP addresses persistent even when endpoints change location—this instead requires frequent updates to routing tables, and reduces the ability to effectively aggregate addresses in routing tables, which reduces scalability.

4.3.6.2 Scalability Issues

The scalability issues are caused by the fact that routing depends on address aggregation, that address aggregation requires alignment of addresses to the network to be effective, which in turns requires rebinding addresses to PoAs to adapt to network changes. This, as we have just seen, leads to connection failures and no guarantees for recovery, resulting in service disruptions. Furthermore, users assume that allocated public IP ranges are permanent and persistent. Therefore networks as a whole are rarely renumbered to improve aggregation. The ‘solution’ to the mobility issue involving letting IP addresses travel with the endpoint, further deteriorating aggregation.

As we shall see next, it is not feasible to change the routing scheme to something more efficient or mobile-friendly.

4.3.6.3 Flexibility

TCP/IP is inflexible in the sense that many kinds of changes affect the APIs and would break applications. In particular, neither protocols, name resolution system (NRS), nor routing can change significantly without syntactical or semantic changes to the API.

Most changes to protocols would affect the socket API. This is because the socket API is so tightly involved with protocol semantics and management, and because applications actively choose which protocol to use.

Updates to the NRS, for example a change to or away from DNS, would require updates to the part of the API involved in DNS lookups as well as how the results should be used with the socket API. This is due to addresses being exposed via the socket API, and the choice that applications should manage name resolution themselves.

Routing changes that require new address formats would change the address data formats and associated semantics of the socket API.

4.3.6.4 Security

Any socket in TCP/IP can connect to any other socket, as long as the EID of the latter is known to the former. The EID is easy to find out or guess. The IP can be looked up using DNS or using network knowledge readily available. The protocol number is standardised

and globally agreed. Furthermore, well-known port numbers makes for good initial attack vectors, and the limit of approximately 2^{16} makes port scanning feasible.

Source address field spoofing makes many denial-of-service (DoS) attacks simple. In general, any attacker can spoof any protocol field if it gains an advantage—there is no identity authentication methods at the TCP or IP level (unless IP Security (IPsec) is used).

IPsec IPsec (Kent and Seo 2005) provides authentication and authorisation of both headers and payloads, including the sending and receiving IPs. It provides integrity and confidentiality of packets. It is a L3 protocol separate from IP, and require compatible routers and hosts. IPsec depends on Internet Key Exchange (IKE) (Kaufman et al. 2014) to set up the shared state required. Firewalls do not need to be IPsec-aware other than they will need to be configured to let IPsec packets pass and they will need to allow the initial setup by IKE, requiring them to allow passing of UDP packets on certain ports—but they do not need an IPsec implementation, as hosts or routers do.

While IPsec provides security as described above, it does not address the root causes described in the next section, and in particular it does not solve TCP/IP mobility issues.

4.3.6.5 Root Causes

In summary, the following choices in TCP/IP are the most important root causes of the issues listed in the previous sections:

1. Letting the PM, manifested as ‘a socket’, be the interaction point between applications and the network service. Applications therefore need to handle recovery when PMs are invalidated. If PMs were managed in the network stack, it could handle recovery itself.
2. Letting EIDs for the PMs be composed of specific network-identifiers. If these identifiers change, the EIDs and by that the PMs are invalidated.
3. Letting EIDs be exposed in the API directly. This means that the EID format cannot change without changing the API and all applications that depend on it.

Together, they mean that when PMs are invalidated, applications need to handle recovery *and* that PMs are invalidated when addresses change or ports change *and* that none of this can be fixed without breaking the API and the applications dependent on it, which are almost all.

Mobility and network renumbering is also impeded. With only limited network renumbering, it is harder to topologically re-align addresses with a changing network graph, which

increases address fragmentation and hinders address aggregation, aggravating routing table scalability. Furthermore, the IP-EID rigidity creates an incentive to facilitate mobility by a more dynamic IP-PoA mapping in routing tables, further amplifying the aggregation issues.

It is also worth noticing that these routing scalability issues could be handled, if networks could be renumbered or address formats could be changed without affecting applications.

4.4 Discussion

The Internet is a tremendous success and has become the epitome of the information age. Its conception and deployment at a time with very limited computing resources and without prior experience is a huge accomplishment. Getting a distributed system at that scale to work effectively at that time (and probably even today) was close to miraculous and is a testament to the competence of its designers.

However, its design is characterised by a focus on getting things to work in the short term rather than on deriving and specifying the right solution for the long term. This mindset is best captured in the IETF mantra: ‘We believe in rough consensus and running code’ from (Hoffman and Harris 2006).

In Section 4.3, we identified several issues with the TCP/IP architecture and their root causes. Here we argue that these root causes stem from ignoring some of the design principles we outlined in Section 2.3.

Figure 4.4 show the correlation between applied principles (or lack thereof) and properties (or lack thereof) of the Internet architecture.

The legend in Figure 4.5 shows how its correlations should be interpreted. For example, a lack of generality (one of the four principles at the bottom of Figure 4.4) leads to special-purpose protocols and layers (middle), which leads to a lack of simplicity (top)—here following a single, simple path through the figure. The ‘lack of’ is shown in the figure with a white diamond or white arrowhead, as explained in Figure 4.5. Another example is the choice of letting IPs being tied to the network graph, which leads to the possibility of aggregation, which improves scalability (shown with a black arrowhead).

In sections Sections 4.4.1.1 to 4.4.1.5 we break down Figure 4.4 and in each section provide a figure only including the paths relevant for each property discussed.

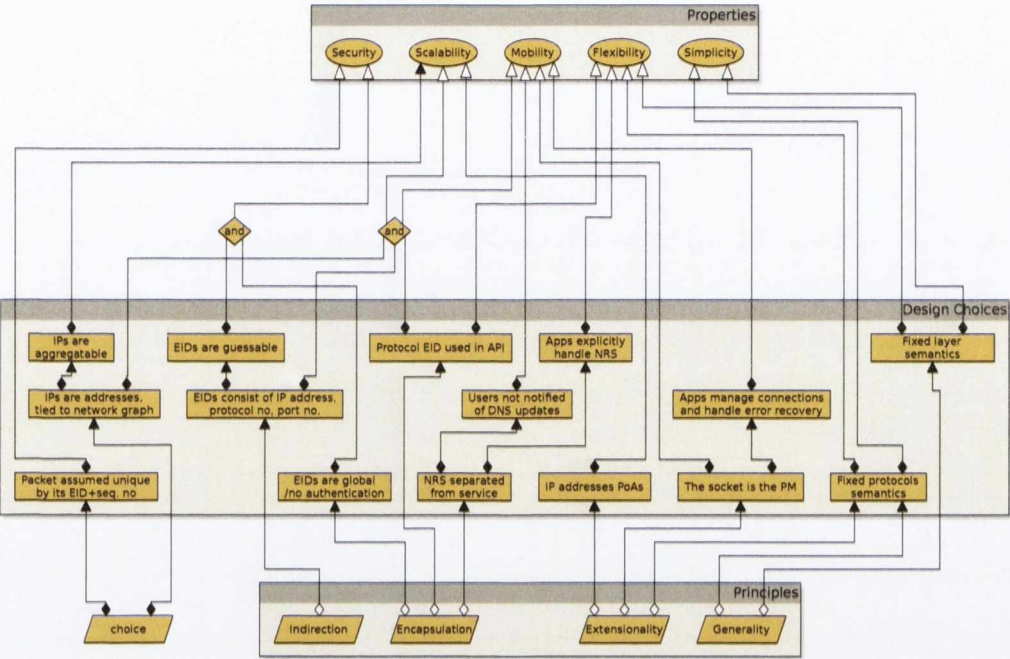


Figure 4.4: Principles applied or not applied to the Internet architecture, and the consequences thereof.

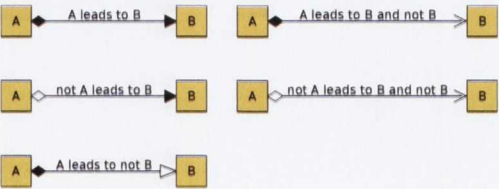


Figure 4.5: Legend for Figures 4.4, 4.6, 4.7 and 4.9 to 4.11.

4.4.1 Properties

Here we discuss how properties (or lack thereof) of TCP/IP can be back-tracked to principles (or lack thereof).

A note on terminology: we distinguish between ‘flows’ and ‘connections’. A flow is the data-stream and the associated state of it between application entities (AEs). A connection is the stream and associated state between PMs. In TCP/IP, these terms are one and the same, because an application binds to a socket, which is the PM of (for example) the TCP stream. Therefore, in TCP/IP, the application is tightly bound to the protocol and state of the underlying connection. As we see in Chapter 5, RINA introduces a level of indirection between the AE and the PM. This decouples the application from the underlying protocols, providing a way to solve many of the problems that TCP/IP suffers from.

4.4.1.1 Flexibility

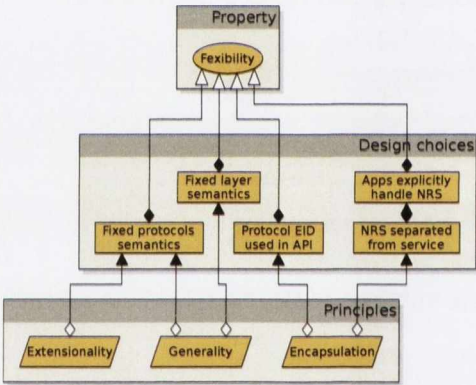


Figure 4.6: Flexibility property of the Internet

Fixed Protocol Semantics The Internet is built on TCP/IP (Postel 1980a,b, 1981a,b) and UDP (Postel 1980c). V. Cerf et al. 1974; Postel 1980b, 1981b specifies an API based on the concept of a *socket*, identified via its IP address and port number. The prevalent implementation is the 1983 BSD4.2 socket API (Leffler et al. 1983).

In the socket API, an application chooses its socket semantics by specifying a combination of *domain*, *type*, and *protocol*, where the latter can be left out if the two former is enough to uniquely identify the protocol of the underlying infrastructure. The socket types available for the Internet domain (AF_INET) were originally either SOCK_STREAM for stream-based, ordered and reliable semantics or SOCK_DGRAM for datagram-based, unordered and unreliable semantics.

These types reflect the semantics of the underlying protocols, TCP and UDP, respectively. Because the semantics of the protocols are reflected in the API, and the applications depend on these semantics, changing the semantics of the protocols is impossible without breaking existing applications. Adding new protocols is possible, for example the Stream Control Transmission Protocol (SCTP) (Stewart 2007) and the Lightweight User Datagram Protocol (UDP-Lite) (Larzon et al. 2004) have seen some deployment and use, but because they have different semantics compared to existing protocols, applications need to be tailored specifically for them and explicitly choose to use them. Because new protocols are not supported everywhere, many applications revert to depend on TCP and UDP only (Day 2016), even when other protocols would be more appropriate.

The socket API's exposure of protocol semantics makes the architecture inflexible with regards to changing the semantics of existing protocols or adding new protocols.

Because a PM is tightly bound to its protocol, and the socket is a manifestation of a PM, the socket API is highly dependent on the protocol that the socket drives. For example, the API controlling a TCP socket differs from that controlling an UDP socket, because the two protocols have different semantics that require different APIs to manage. The AP chooses the protocol (by specifying a valid combination of the *domain*, *type* and, optionally, *protocol* parameters to the *open* operation) and by that, the API. If protocols change in a way that affects the PM API, or if different protocols are desired, the AP needs updating.

The address format cannot change without extensive changes to APIs and applications, and without requiring global coordination, so addresses in other formats cannot in practice be used, even if they are more appropriate for aggregation.

Protocol EIDs In API The socket API uses EIDs from TCP/IP to identify the sockets, namely the three-tuple consisting of an IP address, a protocol number, and a port number. Therefore, EIDs of the protocols cannot change without it affecting the APIs. This would break existing applications.

Because IP addresses are contained in EIDs, the IP address format is locked too. Therefore, we cannot change routing in any way that would require changes to the addressing formats.

In general, any change requiring changes to the EID is made exceedingly difficult. For example, for the same reason, we cannot introduce named ports or a larger range of ports, and it is very difficult to change the tight integration and dependencies between TCP and IP.

The API exposes the EID with its inherent semantics, because it is used as socket identifier. If these semantics change, applications need to change too. Therefore, any change

cumbersome because it requires global agreement. This is the reason that port numbers, protocols numbers, and IP formats and semantics needs global consensus. The still ongoing change from IPv4 to IPv6 is one example of how this is problematic.

Explicit NRS Applications need to find out the EIDs used for the socket API operations. For this, DNS and various rewriting rules are used via the `getaddrinfo` POSIX API (Gilligan et al. 1999). While the API abstracts certain details—it is, for example, designed to be used with the socket API in ways that makes the application oblivious to whether or not IPv4 or IPv6 is used—the fact remains that its output is tied to the input of the socket API, and that semantically name resolution is managed explicitly by the application before socket calls, and not, for example, transparently by the OS, encapsulated in the IPC service provided by the socket API.

Due to the explicit name resolution process and API and its dependency with the socket API, the NRS—including DNS—cannot be changed without major modifications. The fact and the way that name resolution is handled by the application is difficult to change too.

Name resolution is managed by the application. If the name resolution service changes, for example if its input or output format changes, the applications must change too.

Fixed Layers The socket API is an implementation of a user-API specified in V. Cerf et al. 1974. The same document states that there is no interface to lower layers.

This means that layer integration is tailored specially for the two specific layers that need to interact and is, specific to the data formats and implementation details of both layers. Combining layers in new ways, or introducing new layers, is difficult and complex due to these integration issues.

Compositions of layers other than the standard five-layer composition goes against the grain of the model. That there is a need to compose layers in new ways is evident when we look at the typical layering in core networks where the number of layers is ten or more. Confusion arises when trying to talk about ‘layer 3’ in a scenario where there are multiple IP layers; or when a new protocol such as MPLS enters the field and is described as belonging to ‘layer 2.5’. These are symptoms that the model diverges from real world needs.

4.4.1.2 Mobility

EIDs Depend on Network The EIDs used for socket identification contains the IP address and port number, which describes a network path to the socket.

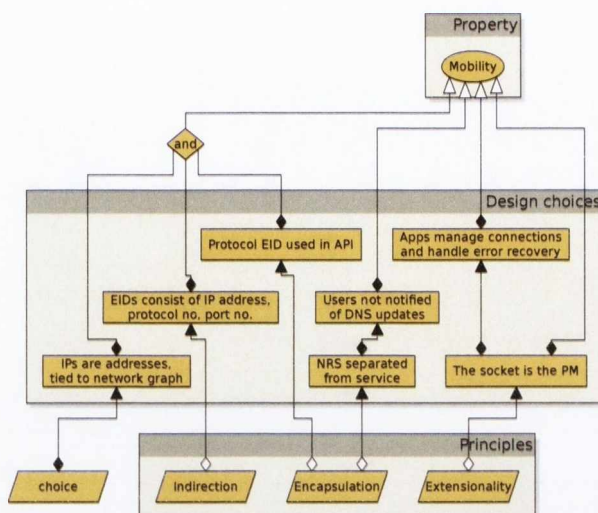


Figure 4.7: Mobility property of the Internet

If the actual network path changes, the EID no longer specifies how to reach the socket, the socket is invalidated, and the connection breaks. This happens if the IP address changes, for example caused by a change of PoA, e.g. when the host or process move in the network, or if a network is renumbered.

Therefore, mobility causes connections to break, requiring applications to recover, if at all possible.

Renumbering PoAs invalidates all connections using those PoAs. Applications must handle the connection failure and additionally detect—e.g., by a prolonged timeout—that another application’s address has changed, before probing DNS for an updated address.

The socket is bound via the IP in the EID to a particular PoA. Therefore, a socket cannot change PoA. If the PoA changes, for example if the PS or whole network is renumbered, for example due to a change of location; or if a new EID is needed because of sequence number roll-over, the socket becomes invalidated. As we saw in Section 4.3.1, connections are managed by the AP. Therefore, the invalidation have to be detected and handled by the AP.

The Socket is both the Service Endpoint and the PM A PM contains the protocol state of one end of a connection, as described in Section 2.4.1. The connection state is the combined state of its PMs. In TCP/IP, the PM is manifested by the *socket*.

The socket exists as a data structure in the OS and can be managed by applications via the socket API. The socket has both type and state, shown in Figure 4.8.



Figure 4.8: TCP/IP socket states. For the two types of sockets—TCP or UDP—the possible states are: *u*—the unbound state, where the socket is created but has not been assigned an EID yet; *b*—the bound state, where an EID has been assigned to it, but no peer is bound yet; *a*—the associated state, where the socket is bound to its own EID and associated with its apposite’s EID and state (only relevant for TCP/IP); *t*—the terminated state, after the socket is closed.

A socket is identified in the OS via its EID or via a pair of EIDs, depending on its type and state: A TCP socket in state *a* is identified via the pair of EID of both ends’ PM; a socket in state *b* is identified solely by its own EID.

From the state diagrams in Figure 4.8, we see that once a socket is bound, it cannot be unbound, and once a TCP socket is connected to an apposite, it cannot be disconnected or reconnected to another. Instead, a new socket must be created with the new binding and possibly association.

At the same time, sockets are the service endpoints from the point of view of the user of the service, i.e., the application. Applications are expected to manage sockets, and by that connections, directly via the socket API. Because the socket is managed by the application, errors and recovery also need to be handled by the application, even though network conditions might be the cause of the error. Broken connections are therefore a problem for the applications.

Because applications manage individual connections explicitly, and because the EIDs dictates the path of the data flow of the connection, there can be no multihoming. Without multihoming, mobility becomes cumbersome.

The socket is bound via the IP in the EID to a particular PoA. Therefore, a socket cannot change PoA. If the PoA changes, for example if the PS or whole network is renumbered,

for example due to a change of location; or if a new EID is needed because of sequence number roll-over, the socket becomes invalidated. As we saw in Section 4.3.1, connections are managed by the AP. Therefore, the invalidation have to be detected and handled by the AP.

A socket is statically bound one-to-one with a handle in the AP, implemented with a file descriptor. Once a handle is created it is fixed to that particular socket for the remainder of its life time.¹ If the socket becomes invalidated for any reason, the AP will need to close it and open a new one.

NRS separated from Service Because applications decide when to interact with the NRS and there is no way for the NRS to initiate an interaction with the applications; applications cannot be notified when mappings in the NRS change. Therefore, each application needs to decide when to look up names and how long answers should be cached for.

4.4.1.3 Scalability

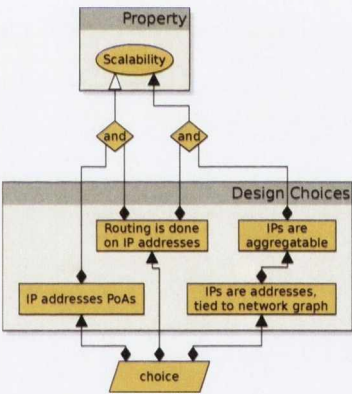


Figure 4.9: Scalability property of the Internet

Routing on PoAs TCP/IP route on IPs which identify PoAs. Routing table sizes are therefore bounded by the number of PoA.

For a network with N nodes, E edges, and average degree $d = 2E/N$, there are $P = 2E = dN$ PoAs.

Data from Siganos 2006 suggests a constant average degree of the Internet graph of around 4. This suggests that the upper bound on routing table sizes is a factor 4 larger if we route on PoAs than if we route on hosts.

¹The handle may be duplicated and shared, but will refer to the same socket.

In general, $2(N - 1) \leq P \leq N(N - 1)$, where the lower bound is for an acyclic graph and the upper bound is for a fully connected graph. If the average degree increases as N increases, the routing table size growth is superlinear.

Aggregation To prevent routing tables from growing linearly or superlinearly with N , which is problematic for an exponentially growing Internet, IP addresses are aggregated.

Aggregation requires that the IP addresses are aligned with the network graph. This has two consequences: First, if IP addresses are mobile, they are not aligned with the network and cannot be aggregated. Secondly, given that the Internet graph is not static, IP renumbering is necessary occasionally to realign IP addresses with the network graph. As noted in Kurose and Ross 2013 renumbering requires reconfiguration of the entire connection and thus breaks existing connections, and relies on application error recovery. Therefore, such renumbering is cumbersome and therefore avoided.

Routing table entries can be aggregated if the topology of the address-space correlates to that of the network graph, i.e., if entities that are near each other in the network graph have addresses that are near each other in the address space (for some definition of nearness).

Keeping the addresses aligned with the network graph to allow for effective aggregation is difficult because renumbering disturbs users of the network and might lead to application failures.

4.4.1.4 Security

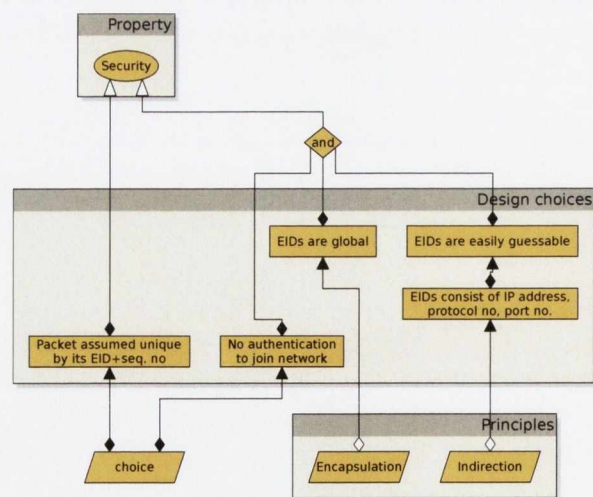


Figure 4.10: Security property of the Internet

Discoverable, Global and Accessible EIDs On the Internet, if you know the EID of a socket, you can send packets to it. There is no prior authentication. While some scoping is provided by network address translation (NAT) and firewalls in practice, all EIDs are global in the basic, unabridged TCP/IP architecture.

Most services rely on well-known port numbers because the prevalent NRS, namely DNS, traditionally only provide FQDN to IP addresses translation.² The protocol number takes only a few values in practice. Both can therefore be guessed by a brute force attack.

The IP part can be looked up by everyone in DNS or discovered from network configuration or traffic interception.

The result is that EIDs are directly discoverable, sockets are in principle globally reachable, and no one is excluded from sending PDUs.

This has serious security implications and has motivated the widespread use of NATs and firewalls. Neither solve the authentication problem, however, they can only be used as a crude filter for obviously unwanted traffic, but they can never block all unwanted traffic; authentication is the only way to provide for that.

Sequence Wrap-Around, EIDs, and Error Recovery A TCP PDU is uniquely identified with its EID pair and sequence number. The sequence number is a 32 bit integer initialised to a randomly chosen number when the connection is established and functions as a byte counter (Postel 1981b). Because TCP does not handle wrap-around, duplicate sequence numbers may appear when more than 4 GiB have been transferred, which can disturb retransmissions and synchronisation mechanisms (Borman et al. 2014; Gont and Bellovin 2012).

4.4.1.5 Simplicity

In TCP/IP, each layer and each protocol is special-purpose, and tightly connected to the other layers. For example, L3 is understood to run IP. L2 will run Ethernet or WiFi. L4 will run UDP or TCP. Each layer have a selection of possible protocols acceptable for that layer. Each protocol provides a very specific and fixed quality of service (QoS). The multiple special-purpose protocols provides an inflexible selection of QoS while at the same time requiring numerous protocols to be implemented in each layer, reducing the simplicity of the architecture.

Because the layers need to pass PDUs between each other, they need some interface

²In theory, the SRV RR type could provide mappings to complete EIDs, but it has not seen widespread use, except for certain services such as the Session Initiation Protocol (SIP).

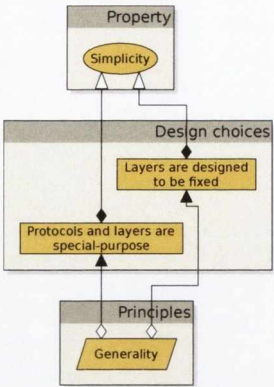


Figure 4.11: Simplicity property of the Internet

between them. For example, a lower layer will need to know to which PM in the layer above an incoming PDU should be passed on to. An Ethernet PM in L2 will need to know whether to pass it to the host’s IPv4 or IPv6 PM in L3, and an IPv4 PM in L3 will need to know whether or not to pass it to the TCP or UDP PM in L4. Beside being able to identify where to pass packets, the exchange between specific protocols of adjacent layers needs to be special-tailored for the combination of protocols: the protocol number (in L3) or ethertype (in L2) identifies which upper-layer component consumes the package, and helps the stack choose the appropriate mechanism. This special-tailoring of interfaces between layers increases complexity.

Fixed Layers Due to the lack of inter-layer APIs, layer composition requires customised glue-code at a low abstraction level, typically using PDU (or PDU pointer) queues. Because protocols differ from layer to layer, headers need to be prepared by the layer that enqueues the PDU to match those of the layer that dequeues them. The enqueueing layer thus need to know the type of the dequeuing layer before the PDU is enqueued. Any new combination of enqueueing and dequeuing layer require new glue code. The glue code requires translation of addresses from one layer to another. This requires a custom mapping service between those two specific layers, including mechanisms to keep such a mapping up to date, e.g. ARP.

Fixed Protocols Each protocol provides a fixed QoS. Ethernet, MPLS, IP, TCP and UDP all provide data transport, but differ in routing mechanisms, scope, degree of support in other layers and applications, and service guarantees. A change in QoS requirements can lead to a change in protocol, requiring comprehensive network and system upgrades. The

one-QoS-one-protocol mindset means that the number of protocols in and surrounding TCP/IP is large.

Notice that the most common TCP/IP transport protocols, namely TCP and UDP, cannot directly support the reliable exchange of messages—if choosing TCP, you get reliability, but you exchange streams and not messages; if choosing UDP you get messaging, but the transport is unreliable. A solution is provided in RINA, where a single transport protocol will always exchange messages of a specific size (which is defined by the sender), but where reliability (and other parameters) are configurable and can be toggled for each flow. Stream-like semantics can be achieved by sending multiple messages and configuring the flow to be ordered. In TCP/IP it would require a change to the stream semantics of TCP, which would require changes in the TCP PMs in the various OS implementation to specifically implement message boundaries—a change which would not be immediately backwards compatible.

4.4.2 The API

Many of the issues of TCP/IP are rooted in the socket API. For example, the choice to use the PM for the socket requires the API to include operations to manage connections and makes its semantics protocol-dependent; and the choice to use the protocol EID for the API EID means that neither can change x affecting the other.

The problem in the first case is a failure to hide the protocol operations from the service API, and in the second case it is a failure to map the data structure defined for the protocol to a separate data structure used for the API. Thus, the principles of encapsulation and indirection were violated in these crucial points.

We posit that a main reason that this happened is that TCP/IP was not designed as a black box service with a clear API. The socket API came after the protocols, and many of the concepts of the API were taken from the well established protocols (Day 2008a; Kurose and Ross 2013). TCP/IP was first and foremost a set of protocols, and only secondly did the designers think about how to interact with these protocols (Tanenbaum and Wetherall 2011).

The consequence is that the API depend on the protocols rather than the other way around. This, in turn, means that the protocols is the central key stone, and therefore they cannot be changed without affecting the APIs. TCP/IP is a set of protocols with an API tailored to this set of protocols, rather than a well-defined abstract service realised by means of suitable protocols (Tanenbaum and Wetherall 2011).

4.4.3 Causing and Recovering From Errors

A general problem in TCP/IP is that applications are expected to react to events in the network, for example, a change of address of a NIC. However, applications do not always handle such events (Kurose and Ross 2013; Tanenbaum and Wetherall 2011). Sometimes they cannot because they do not have the necessary information (for example, they lack a new address necessary to reestablish a broken connection), and sometimes they are not tailored to handle certain kinds of events (e.g., connection broken) because the application developer did not foresee such events (or did not care).

4.5 Extensions of the Modern Internet

The modern Internet is classic Internet with augmentations, specifically IPv6, IPsec, NAT, MIP, SDN, and loc/id, for example ILNP or LISP. The *raison d'être* of the augmentations is to provide the properties listed in our taxonomy as cheaply as possible in terms of money and changes to the classic Internet and its applications.

We do not explain the architecture in great detail as we did with RINA, but instead refer to existing literature. We have included it as a separate architecture to allow an assessment separate from the 'original', traditional classic Internet architecture.

4.5.1 IP version 6

IPv6 (Deering and Hinden 1998) was designed to mitigate IPv4's problem of address exhaustion. IPv6 fits into the TCP/IP model as a replacement for IPv4 in L3. It therefore relies on the applicability and relevance of that model.

IPv6's premise is to solve an isolated problem in IPv4, namely that of address exhaustion. The change to IPv6 requires world-wide changes across layers, both due to the fact that all equipment directly depending on IP, such as routers, need to be upgraded, but also due to the transport layer depending on them for its pseudo-headers necessary to support the choice of EIDs, and the application layer exposing them by using EIDs directly in the socket API. Its adaption is ongoing and slow for these reasons, as well as because IPv4 NAT'ing to a large extent mitigate the address exhaustion problem while additionally providing added security by hiding subnetworks from the larger Internet.

TCP/IP have many problems as outlined in Section 4.3. IPv6 does not solve any of those. In particular, IPv6 does not change the fact that applications are still bound to sockets and need to handle failures and multihoming; that failures are bound to happen due to the

binding of IP to the EID in both protocols and APIs; and that IP fundamentally is bound to the PoA while a more appropriate choice would be the host or the OS.

For a solution that requires such widespread changes, IPv6 solves very few problems.

4.5.2 SDN

Software-Defined Networking (SDN) solves the problem of network management for TCP/IP, i.e. how to configure the network to send which packets where.³ Forwarding has typically been implemented in hardware, and routing information exchanged through distributed routing algorithms using routing protocols such as IS-IS or OSPF. The distributed nature, the hardware implementations, and the lack of a uniform point of entry for network configuration has limited functionality of the management layer. SDN solves the problem by centralising routing computations and enabling dynamic routing and intelligent forwarding.

SDN does not solve the problems of TCP/IP, nor does it attempt to. Its function is only to ease network management and routing configuration.

4.5.3 Summary of Modern Internet Extensions

In the Internet architecture, to change the APIs, and by extension, all applications that uses the APIs, is insurmountable pain. Therefore, the extensions try not to change the APIs, at least not too much.

IPv6, for example, does change the address format, and by that the data structures of the EIDs of the socket API. It does not change anything else: Not the thing the address identifies (it is still PoA), nor the operations or states of the API. Day 2008a points out that the sensible thing to do would be to fix the API while one is changing it in any case, and update the semantics of addresses to identify the process of the PM (which in TCP/IP would be the OS process, which in most cases is equivalent to 'the host'). This did not happen: It was deemed too big a change.

If an endpoint needs to move, at least one of the mappings from the EID to the network PoA needs to change: Either the IP is updated, and by that the EID. Connections then break and need to be reestablished, which is up to the applications. DNS needs to be consulted to find out the new IP, requiring DNS to propagate updates quickly, which is not always the case. This is a brittle solution, with many factors outside of the control of whoever initiated the move. Alternatively, the mapping from IP to PoA can be updated. This requires changes

³SDN defines itself as being in the 'control plane', as opposed to the 'data plane'. We use the definitions of Day 2008a, where the control plane is about error and flow control, and where the management plane is about propagating state that enables the network nodes to do forwarding.

to routing tables, which is traditionally hard to manage, but is completely transparent to applications and DNS. SDN provides an easier way to manage routing table updates and by that the mapping between IPs and PoAs than was previously available. The success of SDN is due to this being done without higher layers knowing or caring.

In all cases, the extensions thrive on the fact that the socket API is fixed. If it were not, the root causes of the problems that these extensions solve could be solved instead.

If the API did not expose the EID and used names instead, address formats and the name resolution system could be changed transparently to applications, and a change like the introduction of IPv6 would require less substantial changes than it did at its introduction. If the API did expose connection management directly, recovery would not have to be handled by applications, and mobility could be supported by making the name-to-address mapping dynamic, rather than making the address-to-PoA mapping dynamic (as SDN does).

4.6 Summary

We have described the classic Internet architecture, how it employs the principles of indirection, encapsulation, extensionality, and generality, and how that affects its properties of flexibility, mobility, scalability, security, and simplicity.

We have discussed how the lack of indirection between flows, connections, and addresses leads to mobility issues. Instead, the architecture's main indirection—that between naming constructs involving FQDNs and IP addresses, facilitated by DNS—happens in applications without guaranteed semantics or performance and with limited support for mobility ((Raghavan, Koponen, Ghodsi, Brajkovic, et al. 2012).

We have discussed how the architecture fails to encapsulate connection management—due to the conflation of flows and connections, both manifested in the socket—which then has to be handled by applications. The architecture also fails to encapsulate addresses, which are exposed via the APIs, making changes to them impossible without changing APIs, which makes the architecture inflexible. Name resolution is also not encapsulated, but happens in applications, which also reduces flexibility.

The lack of encapsulation and the exposure of internal semantics and syntax shows the API's dependency on implementation choices. This specialisation of the APIs and the service to reflect a specific implementation has led to a non-general service and an API that bounds applications to the semantics provided by the backing protocols.

The inflexibility caused by the non-extensional and specialised design prevents the

architecture from evolving—the significant changes that are needed to address its issues will affect the APIs and break existing applications.

Chapter 5

RINA

We present the Recursive Internet Architecture (RINA) (Day 2008a; Day, Bunch, et al. 2013), a clean-slate architecture that provides IPC service. RINA emphasises the employment of the fundamental principles from our taxonomy because they have previously proven successful in (distributed) software design (Clark 1988; Day et al. 2008b) and because the authors do not find good reasons not to employ them. The resulting architecture possess all the desired properties from our taxonomy.

5.1 Premise

Despite the fact that the Internet dominates networking both in teaching, academia and in terms of practical usage, it is known to have several shortcomings hinting that perhaps the things that have now become matters-of-facts should be revisited. Because of this and because of the technical and political polemics experienced during the conception of the Internet and described by the developers of RINA (Day 2008a; Day et al. 2008a; Trouva et al. 2011), they do not accept the dogma of the Internet architecture at face-value and start with a clean slate, where everything is reexamined based on prevalent literature on networking, e.g. Fletcher and Watson 1978; ITU-T 1994; Jain et al. 1988; Pouzin 1973; J. Saltzer 1993; J. H. Saltzer 1978; Watson 1981. This includes naming, protocol design, API design, layers, security, multihoming, mobility, and multicast.

The one invariant in RINA is the service it provides: Because every bit transmitted on a computer network that is not caused by noise or errors has been written by some process, and if it will be read at all, it will be read by a process, RINA is built on the premise that *networking is IPC*.

Instead of first defining the protocols and then decide how applications should use them, RINA seeks to define a general IPC service along with an abstract API. Only secondarily are

protocols and other mechanisms designed to facilitate this service.

5.2 Model

We reiterate the RINA model derived in Day 2008a and specified in Day, Bunch, et al. 2013. We use the third person plural pronoun ('we', 'us', etc.) to retain the flow of the text, but we emphasise that the basis for the work presented in this section is Day's and that we merely present it for the context of this dissertation.

We start with a simple model of two APs communicating directly within a single PS. We then gradually expand the model, until we end up with the full RINA model. The derivation covers the following iterations of increasing complexity:

1. **Local:** Communication between APs on the same PSs.
2. **Link:** Communication between two PSs.
 - (a) One pair of APs communicating at a time.
 - (b) Multiple pairs of APs communicating simultaneously.
3. **Network:** Communication between multiple PSs in a network.
 - (a) Direct communication between PSs.
 - (b) Relayed communication between PSs.
4. **Internetwork:** Communication across multiple networks.

5.2.1 Local

The starting point is the situation where two APs within the same PS need to communicate. The IPC within the PS is handled by an IPC facility provided by the OS.

A part of the AP is directly involved in the communication with the IPC facility. We call this part the AE. The difference between the AP and the AE will become relevant later; in the following, it is not relevant and therefore we ignore it.

The IPC facility provides an API that looks as follows:

```

result ← allocate(? → dst, ? → pid, qos)
result ← send(pid, buf)
result ← receive(pid, ? → buf)
result ← deallocate(pid)

```

All operations return a result with information about the success or failure of the operation. Some operations have parameters whose value might be updated after the operation returns;

these are prefixed with ' $? \rightarrow$ '. The `allocate` operation allocates resources to communicate with a destination AP given by *dst*, with communication QoS parameters specified in *qos*. The source is implicitly understood by the IPC facility to be the calling AP. A port identifier (PID) is created by the IPC facility and saved in *pid*, which can be used by the AP for subsequent `send` and `receive` operations for this communication instance. When the communication instance is no longer needed, each side can invoke `deallocate` to free resources; this will also free (and invalidate) the PID.

If no destination is specified, the `allocate` operation will block and wait for an incoming allocation attempt from another AP. When this happens, the name of the connecting application will be saved in *dst*. AP names are unique within the PS and can be shared across applications and the IPC facility. We cover AP naming at a later point. Port identifiers are only unique between the IPC facility and the AP, and should not be shared across applications. Buffers are assumed to include information about their size. The buffer boundary is preserved across the communication; an application invoking a `send` operation with a certain buffer size will define the buffer size returned by the receiving application's corresponding `receive` operation.

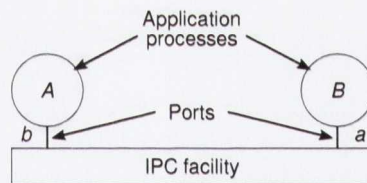


Figure 5.1: Local communication between applications **A** and **B**. Adapted from Day 2008a, Figure 6–4.

Let us walk through a simple communication scenario between applications **A** and **B** as depicted in Figure 5.1:

1. **A** invokes `allocate` to start communicating with **B**. **A** need to know the name **B** a priori. **A** also specifies the desired communication properties *p*.

A : $\dots \leftarrow \text{allocate}(\mathbf{B}, ? \rightarrow \text{pid}, p)$

The IPC facility allocates resources to this communication endpoint instance, and assigns a PID *b* to it.

2. The IPC facility finds **B** and determine whether or not **A** has access to **B** using the search rules and access control mechanisms, respectively, of the OS. If **B** is not already

running, the IPC facility might cause **B** to be instantiated. In any case, **B** should eventually be blocked on a listening allocate operation:

B : $\dots \leftarrow \text{allocate}(\text{?} \rightarrow \text{dst}, \text{?} \rightarrow \text{pid}, p)$

3. The IPC facility will allocate resources for **B**'s communication endpoint instance, and assign a PID **a**. It then notifies **B** by unblocking it and updating the output variables *dst* and *pid*:

B : $\text{result} \leftarrow \text{allocate}(\text{A} \rightarrow \text{dst}, \text{b} \rightarrow \text{pid}, p)$

4. The IPC facility now notifies and unblocks **A** similarly, passing back the PID **a** in the process:

A : $\text{result} \leftarrow \text{allocate}(\text{B}, \text{a} \rightarrow \text{pid}, p)$

5. **A** may now send data to **B** by invoking:

A : $\text{result} \leftarrow \text{send}(\text{b}, \text{data})$

which **B** receives by invoking:

B : $\text{result} \leftarrow \text{receive}(\text{a}, \text{data} \rightarrow \text{buf})$

B can send data to **A** similarly:

B : $\text{result} \leftarrow \text{send}(\text{a}, \text{data}')$

A : $\text{result} \leftarrow \text{receive}(\text{b}, \text{data}' \rightarrow \text{buf})$

6. When they are done exchanging data, they can each deallocate communication resources with:

A : $\text{result} \leftarrow \text{deallocate}(\text{b})$

B : $\text{result} \leftarrow \text{deallocate}(\text{a})$

Multiple allocate operations will yield multiple communication endpoint instances with different PIDs. The IPC facility manages the association between port pairs in order to be able to transfer the data, which happens by means of copying data from one AP's memory space to the other's. We assume this is done reliably and in order; there is little reason not to when the IPC is contained within a single system. Flow control is handled by the IPC facility:

When message queues are full, additional `send` operations block; when they are empty, receive operations block. Errors are caused by either memory leaks or data corruption. The former we consider bugs to be fixed, while the latter are similar to memory corruption (because data messages never leaves the memory), which are handled behind the scenes in modern computers (e.g., using error correcting coding Tanenbaum and Wetherall 2011).

Thus, in this initial model where the IPC and processes are contained within a single system, an IPC facility implemented as a single process can provide reliable, ordered data transfer with error and flow control because there is no network to where packets can get corrupted or lost.

In summary, we have gone through the following steps:

1. Source allocation initiation
 - 1.1. Source invokes `allocate`.
 - 1.2. IPC facility allocates resources for the source endpoint.
2. Destination search, access and readiness
 - 2.1. IPC facility searches for the destination and determines if the source has access to it.
 - 2.2. IPC facility checks if the destination AP is running and ready. If the AP is not running, it might start an instance, waiting for destination to become ready. The destination is ready when blocking on a listening `allocate` call.
3. Destination allocation
 - 3.1. IPC facility allocates resources for the destination endpoint.
 - 3.2. IPC facility notifies destination AP which unblocks its `allocate` call.
4. Source allocation finalisation
 - 4.1. IPC facility notifies source AP which unblocks its `allocate` call.
5. Data exchange
 - 5.1. The APs exchange data via the IPC facility using `send` and `receive` operations.
6. Deallocation
 - 6.1. Both APs invoke `deallocate`.

5.2.2 Link

Now we expand our model and consider applications communicating across two systems. We imagine the two systems are connected via some physical medium, for example a wire. Abstractly, the IPC facility covers both systems via the medium; we therefore call it a distributed IPC facility (DIF). A DIF spanning two systems will consist of two loosely

coupled (distributed) IPC processes (DIPs) communicating over the wire. These DIPs are ‘lower-level’ APs within each system. For applications, there is not much change; the real change is in what happens inside the DIF. The situation is depicted in Figure 5.2.

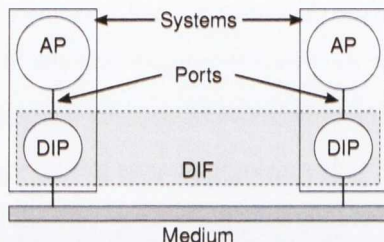


Figure 5.2: Applications communicating between two systems

We go through the same six main steps as previously and discuss any needed changes as we go along. Like before, we assume that AP *A* initiates the communication with another AP *B*:

1. Source allocation initiation
 - 1.1. Source invokes `allocate`: $A : \dots \leftarrow \text{allocate}(B, ? \rightarrow pid, p)$
 - 1.2. The local DIP allocates resources for the source endpoint at *A*.

The first change from the previous scenario is that the application name space will need to span both PSs; it is no longer confined to only one PS. We make application names independent of the system they run on so that applications may change systems without also changing names.

2. Destination search, access and readiness
 - 2.1. The DIF searches for destination *B* and, if found, determines if the source has access to it.
 - 2.2. The DIF checks if the destination AP is running and ready. If the AP is not running, it might start an instance, waiting for destination to become ready. The destination is ready when blocking on a listening `allocate` call.

In order to find *B*, the DIF needs to search both systems. If *B* is not found locally at the same system as *A*, it must be on the other system or not exist at all. Thus, the DIP local to *A* need a way to ask the remote DIP if *B* exists there and if *A* is allowed access to it. To facilitate this, we introduce the IPC access protocol (IAP) capable of exchanging the necessary information between the DIPs. To manage resources, we introduce an IPC management task running inside each DIP. It handles IAP requests and also interfaces with the local OS.

Authentication is not a part of IAP, but is delegated to the applications because in the

end only the application can make decisions about who to authenticate. This means that the IPC will allow access based on the best knowledge available to the DIF. The validity of that knowledge will be confirmed once the applications themselves authenticate each other.

Before IAP messages can be exchanged, the DIPs themselves need to authenticate each other. This is called *enrolment*, which is explained in Section 5.2.6.1. For now, we assume that enrolment has taken place, and that the two DIPs constituting our DIF have authenticated each other.

Because data is no longer transferred using shared memory, but over a wire, errors, synchronisation, reliability and flow control need explicit handling. To do this, we introduce the Error and Flow Control Protocol (EFCP), which we discuss in detail in Section 5.2.5. The IAP is dependent on the EFCP, as is any other communication between the two DIP. The EFCP hands messages to a driver, which is responsible for transmitting them on the wire.

With the IAP, the EFCP and the IPC management task in place, we can elaborate on point 2 from above:

2. Destination search, access and readiness

2.1. The DIF searches for destination **B** and, if found, determines if the source has access to it. First the DIF lets the DIP at the local system search for it according to the local OS's search rules. If **B** is not found locally, the DIP needs to send an IAP message to the remote DIP. To do this, the following needs to happen:

2.1.1. An EFCP communication instance is created with properties suitable for a DIF-internal DIP-to-DIP connection.

2.1.2. The IAP flow is encapsulated in the EFCP, and an IAP connection is established between the IPC management tasks of the two DIPs, who authenticate each other before continuing.

2.1.3. Using IAP, the local DIP requests the remote DIP to search for **B** and determine if **A** has access to it. The remote DIP does this using its OS's search and access control rules, and responds to the requesting DIP with the results.

2.2. The DIF checks if the destination AP is running and ready. If the AP is not running, it might start an instance, waiting for destination to become ready. The destination is ready when blocking on a listening `allocate` call.

We continue with an elaboration of the rest of the steps incorporating the inclusion of the IAP and EFCP as discussed above:

3. Destination allocation

3.1. Via IAP to the remote DIP, the DIF allocates resources for the destination endpoint

- b) There are only two systems and therefore only two DIPs and therefore only two EFCP PMs.

Therefore, the EFCP PM can only hand the outgoing EFCP PDU to the driver, which has only one medium on which to put the PDU, leading to exactly one other system with one driver receiving it, who again has only one DIP with one EFCP PM to receive it.

- c) There is only one AE per system bound to the DIP at a time.

Therefore, once a PDU has been received by the EFCP PM, there is no ambiguity as to which AE the SDU should be handed off to.

So, the source and destination of a EFCP PDU is always unambiguous, and PDUs therefore do not need destination or source identifiers in the PCI in this simple link model.

5.2.2.1 Link with Multiple Connections

One connection is between exactly one pair of EFCP PMs. To allow multiple simultaneous connections, we need multiple pairs of EFCP PMs. To distinguish one EFCP PM from another, we introduce the EID. EIDs have to be unique for each EFCP PM within the DIP. For now, we use the PIDs as EIDs.

A connection is then identified by a connection identifier (CID) consisting of the EIDs of its EFCP PMs. Because the EIDs are unique inside each DIP, the CIDs are unique across the two communicating DIPs and can therefore uniquely identify a connection. Using PIDs as EIDs preserves this property.

We still do not need addresses, because there are only two PSs in our model so far.

Now multiple EFCP PMs are competing over the medium connecting the two PSs. we need a multiplexing task (mux) with the following responsibilities:

1. Schedule queueing, i.e. which EFCP PMs gets to use the medium and when.
2. Manage queue lengths.
3. Pass PDUs between the medium and the EFCP PMs.
4. Concatenate or split PDUs if beneficial.

The QoS parameter passed to the allocate operation affects both how the mux balances the use of the medium amongst the connections as well as how each individual EFCP PMs behaves. For example, QoS requirements regarding throughput or jitter will affect the mux, while flow control or reliability parameters will affect the individual EFCP PM.

Both ends will need to agree on a consistent policy both with regards to the mux and the individual EFCP PMs. Therefore, the IPC management task at each end controls the

behaviour and allocation of EFCP PMs and the mux and coordinate using the IAP.

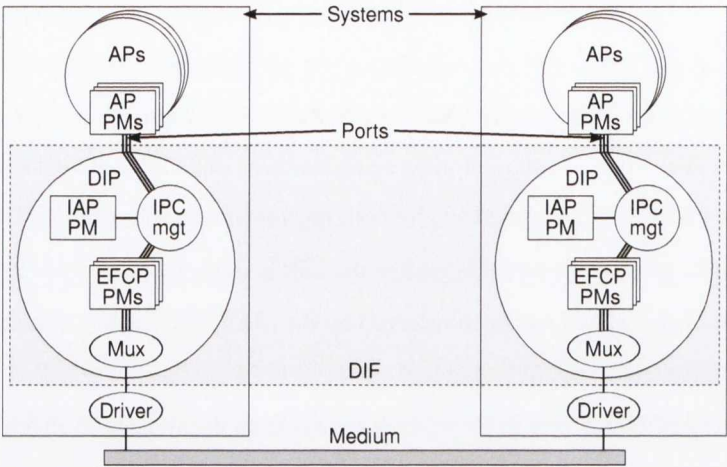


Figure 5.4: Multiple simultaneous connections between two systems requiring multiple EFCP PM instances and a mux in each DIP

The picture so far is shown in Figure 5.4. We now look into expanding the model to include more than two systems.

5.2.3 Network

We now assume a fully connected network of N systems as shown in Figure 5.5. This has two immediate consequences: i) Finding a destination application requires a search across all the systems, and ii) Each system has more than one medium to send data to and receive data from.

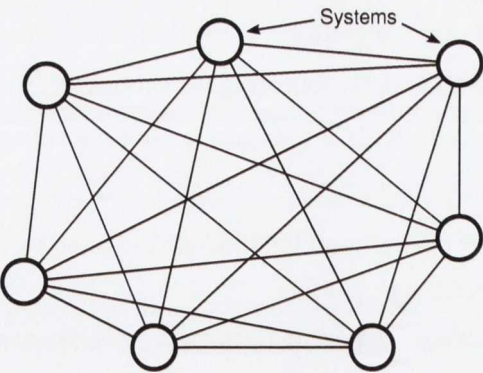


Figure 5.5: N directly connected systems

To tackle the first issue, we introduce a directory: each DIP of the N -system DIF will have a directory where application names and their addresses are stored. we clarify what we

mean with 'address' shortly. The directories of the DIPs exchange information using a new application protocol which we call the resource information exchange protocol (RIEP). RIEP is an instance of the common distributed application protocol (CDAP), which we describe in Section 5.2.5.2.

Because all N systems are directly connected, they will each have $N - 1$ interfaces to usable mediums. A destination AP, if it exists, is always directly reachable from any system by choosing the medium from the source system that connects it to the destination system. In this fully connected model, each DIP's directory will send the set of known local AP names to its neighbours using RIEP, and receive similar data back. The receiving DIP map the received AP names to the medium from which the RIEP update was received. Each local directory therefore maps application names to an internal identifier of the medium at which the application is reachable. An 'address' is therefore such an internal identifier for a medium. It is unique only within the DIP, and it is never shared outside of the DIP; it is not necessary.

To handle the second issue, the IPC management task in each DIP needs to associate each EFCP PM instance with the correct medium given the information available in the directory. A mux needs to exist for each medium to handle input, output and balancing according to the policies for the instantiated connections that use the medium and according to the type of medium.

For now, we have the situation depicted in Figure 5.6. To manage the resources for a

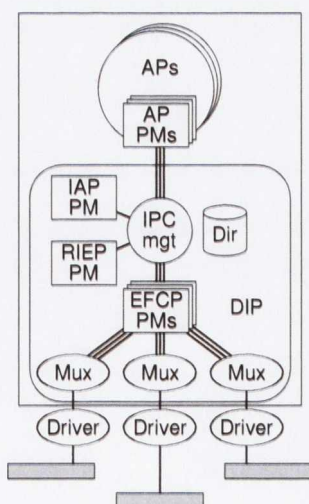


Figure 5.6: An interim picture of a PS in a network of N directly connected systems. We now have multiple mediums, each requiring their own mux to multiplex the PDUs from the EFCP PMs bound to each interface or mux. Adaptation of Day 2008a, Figure 6–11.

specific medium, we introduce a lower-layer per-medium IPC management task. These need to coordinate resource allocation across the medium with their peers as response to requests from the high-level IPC management task. We end up with a repeating pattern as shown in Figure 5.7, where several elements (though not all) of the DIP are repeated on a per-medium basis.

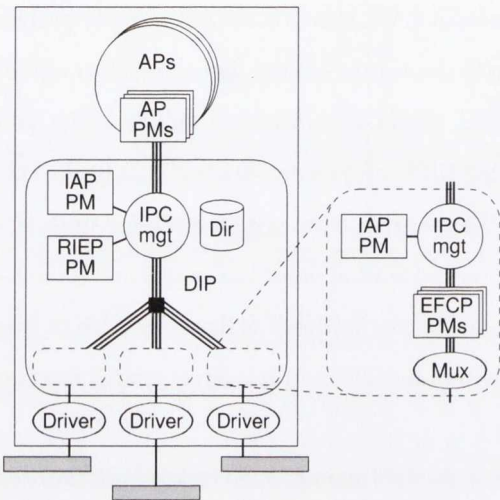


Figure 5.7: Each medium require their own lower-level IPC management task and IAP PM, and we start to see a pattern of elements from the upper-level DIP repeated in a per-medium DIP-like component. Adaption of Day 2008a, Figure 6–12.

A network of N directly connected systems does not scale well as N increases. Now we need to be able to relay communication to be able to reduce the number of direct links.

5.2.3.1 Network with Relayed Connections

We now consider a network of N systems with relayed communication where direct links are unavailable, see Figure 5.8. We distinguish between routers, which are systems that only relay PDUs, and hosts, which act as source or destination for PDUs.

A router connects two or more mediums. It contains a relaying task that forwards PDUs from one medium onto another. A simplified interim simple router is depicted in Figure 5.9, but we expand on this shortly.

Until now, the EFCP PMs of a communication connection have spanned only one physical link because everything has been directly connected. Therefore, we have not had to distinguish between the error and flow handling of the virtual communication connection and that of the physical link. With the introduction of relays, a virtual connection may span multiple physical links across different mediums with different properties. Some error and

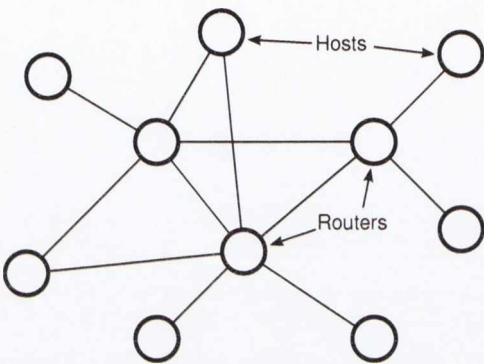


Figure 5.8: N systems indirectly connected. Systems can be categorised into routers, which relay PDUs between systems that are not directly connected, and hosts, which produce and consume PDUs. A system might assume both roles.

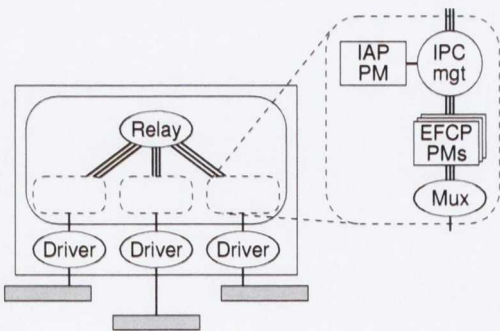


Figure 5.9: A simplified interim router design

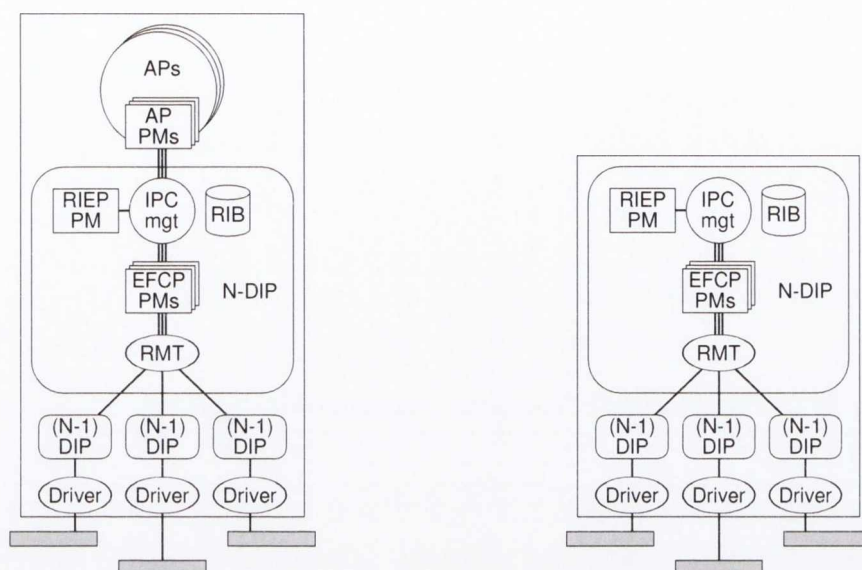
flow control issues are best handled locally across a single medium in order to limit their effects on other systems. Others can be handled at the level of the virtual connection as a whole.

The consequence is that we need to run EFCP across each physical medium as well as for virtual connections. Sometimes, for example, for very reliable mediums or when the application driving the connection has low QoS requirements, the medium EFCP will only be a thin shim, while at other times it is more efficient to let it provide extensible error and flow control. The EFCP policies of different levels can affect each other and must be balanced. We do not cover that here. The important conclusion is that both the virtual and physical levels must provide EFCP.

The hosts look as shown in Figure 5.10a. There are a few important changes: Firstly, it is convenient, if the mux can also handle relaying to provide hosts with bridging functionality. Similarly, it is convenient, if the relaying tasks of the routers can do multiplexing, which allows for concatenating and splitting PDUs in the network where convenient. We call this combined task the relay and multiplexing task (RMT). Secondly, we have absorbed the IAP into the RIEP, and have renamed directory to resource information base (RIB), which includes directory functionality as well as routing information.

We notice how the set of elements repeated for each interface constitute a separate DIP. Thus, the medium is abstracted away as a lower-level (level $(N-1)$) DIF, which is used by the higher-level (level N) DIF in principle with a similar API as the one the top-level application is using with the higher-level DIF. In practice, because both DIFs might reside inside the kernel, the API between DIFs might be different for effectivity.

Each DIP contains all the familiar elements, including a RMT next to the lower-level DIPs. Similarly, we may expand the router shown earlier (Figure 5.9) to take advantage of the recursive pattern, we have just detailed. This is depicted in Figure 5.10b. It includes the RIB, which allows the RMT to make routing decisions, as well as a RIEP PM to be able to exchange routing information with other DIPs in its DIF and to authenticate to be able make the router DIP join the DIF in the first place. The EFCP PMs are needed to facilitate the RIEP, but are otherwise not involved in any relaying functionality. A pure router (i.e., one that has no switching capability (Tanenbaum and Wetherall 2011)) will not have any interfacing applications at the higher level.



(a) A host system. Each medium now has a dedicated $(N-1)$ -DIP which abstracts the medium to look like an $(N-1)$ -DIF.

(b) A router forwarding PDUs between mediums

Figure 5.10: Systems in a relayed network

Addressing

With virtual connections relayed across multiple systems, our destination AP address can no longer be equivalent to the DIP-internal identifier of an outgoing medium (or rather, lower-level DIF), which was how we defined ‘address’ in Section 5.2.3. Instead, we assign unique addresses to all DIPs in a DIF. An address is therefore no longer internal to the DIP but to the DIF. It is not shared across DIFs, for the same reason we did not share addresses as previously defined across DIPs: There is no reason to.

We let the RMT add a PCI containing the source and destination address to all PDU leaving a DIP. The source address becomes known to the RMT when its DIP enroll in the DIF. The destination address can be known, if we maintain an updated mapping between the DIP’s connection instances and destination addresses in the RIB. If an RMT receives a PDU destined for the local DIP, it strips off the address PCI before it passes the PDU to the relevant EFCP PM instance in the DIP.

The mapping to addresses and the prefixing of the relaying PCI containing these addresses happens dynamically when the PDU passes the RMT. This allows connections to persist even if addresses change.

Flows and Connections

Let us start by at this point defining ‘a flow’ as the service provided by the DIF for connecting AEs, and ‘a connection’ for a pair of allocated EFCP PMs sharing state inside the DIF that supports that service.

Until now, we have used the PID as the EID to identify the EFCP PM in a DIP. However, this fixes one specific EFCP PM to a port, makes the CID dependent on the PID, and therefore fixes one connection per flow. We may wish to change CIDs during the lifetime of the flow, for example, when sequence numbers roll over (see Section 5.2.5.1), or we may need several connections for the same flow, for example when the application is permanently multihomed or when its flow needs to be handed over from one lower-level DIF to another, which involves a temporary period where it is connected through several lower-layer DIFs at the same time.

This requires that the PID and the EID are decoupled. We therefore introduce a flow allocator (FA) as part of the IPC management task. During the flow allocation, at each end, the FA creates a flow allocator instance (FAI) for the lifetime of the flow. The FAI provides the mapping between the PID and EIDs for the flow, where an EID uniquely identifies an EFCP PM instance. With its peer, it allocates and deallocates connections for the flow. This is shown in Figure 5.11.

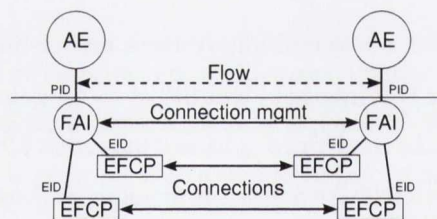


Figure 5.11: Flow and connections between AEs

5.2.4 Internetwork and layers

We now zoom out and picture an N -DIP as a consisting of an RMT and sets of EFCP PMs, as shown in Figure 5.12. The RMT consists of a relay and a mux.

The relay is the AP of one or more $(N-1)$ -DIPs. It maps addresses to either PIDs of $(N-1)$ -DIPs below it, or to ‘the local DIP’, which means the local mux above it. To perform this mapping, the relay is dependent on routing information being updated across the DIF.

The mux connects the relay below it with local EFCP PMs above it. It maps CIDs to EFCP PMs directly, because each EFCP PM is uniquely identified by its EID, which is contained in

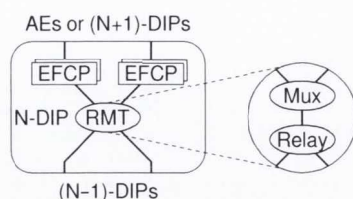


Figure 5.12: High-level picture of a DIP

the CID. The EIDs are allocated by local FAIs and are unique within the DIP.

The EFCP PMs are each mapped to a north-facing port which connects them to their user AEs, which might belong to the relay of an $(N+1)$ -DIP. The mapping from EFCP PMs to port is handled by a FAI for each port.

In Section 5.2.3.1, we noticed a recursive structure where the per-medium physical level $(N-1)$ -DIFs which support the network level N -DIF. In RINA, a ‘layer’ is a DIF, and we use the two terms as synonyms.

A diagram of the layers of a network similar to the simple one shown in Figure 5.8 is shown in Figure 5.13. We notice the following properties of a layer:

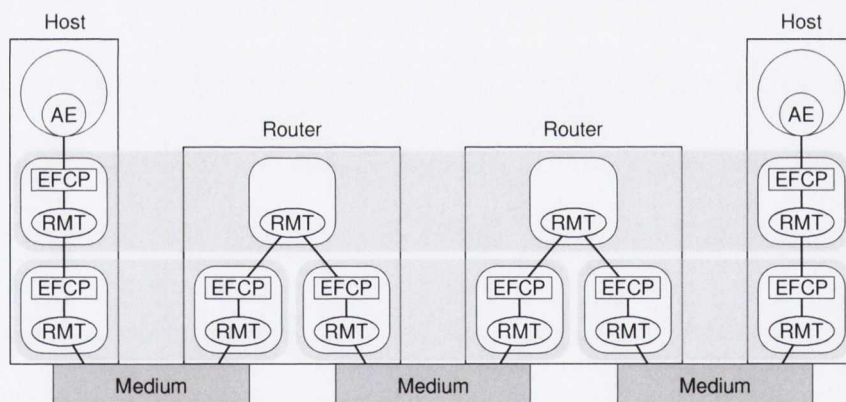


Figure 5.13: DIFs constitute the layers in RINA. Here four DIFs are shown: One network-wide N -DIF, spanning three link $(N-1)$ -DIFs. Each $(N-1)$ -DIF abstracts a physical medium (which we can consider to be a bottom pseudo-layer).

1. A layer's scope is defined by the presence of DIPs of the layer.
2. A layer contains both relaying, multiplexing and error and flow control; however, error and flow control need only be present where flows terminate.
3. Several layers may exist side-by side at the same level.
4. A higher-level layer typically has greater scope than lower-level layers, although one could think of notable exceptions to this, for example, virtual private networks (VPNs).

Layers can be nested as many times as necessary. Figure 5.13 showed an example with two layers for a local network. An internetwork might require an additional layer on top to span several local networks, as well as supporting back-bone layers at lower levels. This is shown in Figure 5.14, where layer *A* connecting hosts 1 and 7 spans networks *B*, *C*, *D*, and where *C* is supported by (what could be) a back-bone layer *F* running over (what could be) fiber links *H* and *I*. Routers can be divided into internal routers (4) or external or gateway

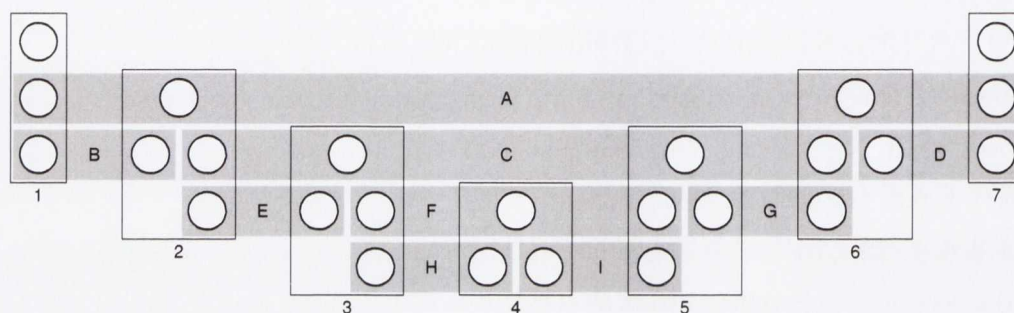


Figure 5.14: More elaborate layer layout of an internetwork

routers (2, 3, 5 and 6), depending on whether or not they connect layers at the same or different levels.

Several things are worth noting:

1. More levels of layers will not in general mean that each system has more layers. For example, in Figure 5.14, no system has more than three levels.
2. Systems can only access systems in the same layer. For example, back-bone systems 3, 4 and 5 are completely inaccessible from hosts 1 and 7 located at the network edge. This is because addresses are internal to each layer: There is no way to address these back-bone systems from the network edge.
3. The lower the layer, the smaller the scope usually is. This means that address name spaces can be smaller at lower layers, which reduces the lengths of address fields, reducing PCI overhead.
4. The lower the layer, the more traffic can be multiplexed on to the same connection—higher-level PDUs can even be concatenated into lower-level SDUs.
5. The lowest layer abstracts the actual physical medium, and will be special-tailored for the medium. The hierarchy of layers thus eventually end up in a concrete physical medium.

The recursive layering, apart from being simple, thus comes with several benefits.

5.2.5 Protocols

To focus on the structure of the recursive model, we have, until now, deliberately avoided details of the protocols in RINA. In this section, we remedy this.

RINA has only two protocols: i) EFCP, which we shall see provide anything from UDP-like best-effort, unreliable, unordered packet delivery to TCP-like reliable and ordered streaming; and ii) CDAP, which is a universal application protocol used to implement RIEP.

5.2.5.1 The Error and Flow Control Protocol

To support a variable range of reliable and unreliable, ordered and unordered, and stream-based and packet-based transfer types with variable formats and sizes of addresses, EFCP is policy-driven. This means that certain mechanisms and protocol data fields are always present and fixed while some mechanisms and fields vary depending on policies. Sometimes policies will eliminate a mechanism altogether.

For example, when addresses are needed, they always refer to source and destination DIPs. These semantics are a fixed part of the addressing mechanisms. However, the format and size of address fields may vary according to policies, and sometimes, for example when a layer does not use relaying and there therefore is no ambiguity with regards to source and destination, they may be left out entirely.

In this section, we cover the following:

- EFCP's natural division into two sub-protocols and how that affects the DIP (Section 5.2.5.1).
- The synchronisation mechanisms' effect on bindings of PMs in the DIPs and the effects this has on resource allocation and mobility (Section 5.2.5.1).
- How configurable policies makes the protocol flexible enough that no other transfer protocols are needed (Section 5.2.5.1).

Nature of Mechanisms and Splitting the Protocol

Day (2008a, Ch. 3) distinguishes between tightly bound mechanisms and loosely bound mechanisms, and the common mechanisms they share.

Common mechanisms are always needed and include delimiting, checksums, relaying and multiplexing. Delimiting makes sure the PM knows where PDUs begin and end, and is implemented with a length field. Checksums allow the PM to detect when data in a PDU has been accidentally changed so that the PDU can be discarded or repaired. It is implemented

with a cyclic redundancy check (CRC) field or a special bit pattern delimiter. The relay of the RMT forwards PDU received from a $(N-1)$ -DIF or from the local multiplexer to another $(N-1)$ -DIF or to the local demultiplexer, depending on its address field. All PDUs received from outside the DIP is error-checked, and faulty ones may be discarded according to policy. The multiplexer schedules PDU from the local EFCP PMs for the relay based on QoS. The demultiplexer hands PDUs from the relay to local EFCP PMs depending on CID and QoS. The RMT is shown in Figure 5.15.

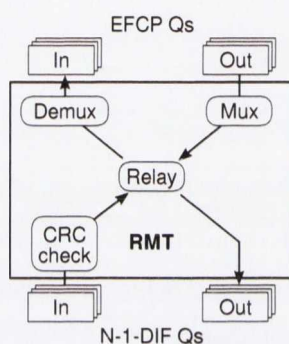


Figure 5.15: The RMT handling error-checking of incoming PDUs, relaying, multiplexing and demultiplexing

Tightly bound mechanisms are directly associated with the data transfer. They include SDU fragmentation, concatenation, ordering and queueing. Fragmentation handles splitting up or reassembling SDUs; concatenation groups several SDUs into one SDU or ungroups them again. This allows SDUs that are too large for some medium to be split up and reassembled at the other end, as well as for grouping several SDUs with the same destination from higher layers into one SDU for efficiency. Ordering and queueing is supported with a sequence number (SN) field for each SDU, so that they can be queued in the correct order when passed to higher or lower layers. None of these functions can be separated from the SDU.

Loosely bound mechanisms are not bound to a specific SDU. They include retransmission and flow control, and synchronisation. This involves managing a state vector for the connection. The state vector keeps track of various timers and the SNs of received PDUs. Retransmission control involves sending acks according to policy when PDUs are received, or retransmitting PDUs when required, again depending on policy and the state vector. Flow control involves updating the sender with information about how many PDUs more it may send, e.g., for a sliding window flow control scheme, or the rate at which it should

	Mechanism type		
	Common	Tightly bound	Loosely bound
Function	General	Data transfer	Data transfer control
Mechanisms	Relaying, multiplexing, delimiting, error check	Fragmentation and reassembly, concatenation and separation, ordering	Retransmission control, flow control, synchronisation
Protocol fields	Addresses, CIDs (including QoS), CRC, PDU type, length/delimiter	Fragmentation flags, SN	Ack SNs, Nack SNs, Right window edge SN, rate, interval
SDU bound	No	Yes	No
PDU types	All	Data	Control: Flow, Ack, Nack, Syn
Time scale	PPT	PPT	RTT
State I/O	None	Write	Read and write

Table 5.1: Common, and tightly and loosely bound mechanisms and their properties

send PDUs, e.g., for a rate-based flow control scheme. Synchronisation deserves separate treatment, and thus is covered in Section 5.2.5.1.

Table 5.1 lists the types of mechanisms and their properties. The functionality of the EFCP falls into two parts: One for the tightly bound data-dependent mechanisms concerned with data transfer, and one for the loosely bound data-independent mechanisms, concerned with data transfer control. The former work over short time scales in the order of PDU processing time (PPT) and its functions does not depend on any other state than that which is contained within each PDU. The latter work over longer time scales, in the order of round-trip time (RTT), and depends on and updates state that needs to survive between handling each individual PDU.

This leads to a split of the EFCP PM into two sub-PM: A the Data Transfer Protocol (DTP) PM and a the Data Transfer Control Protocol (DTCP) PM, sharing a state vector. This is depicted in Figure 5.16. Protocol data fields are grouped into different PDU types, which are handled by one of the two PMs.

This arrangement into two sub-PM and associated PDU types and the fact they are as decoupled as is the case, has the advantage that if policies do not mandate a certain function, its mechanism, associated PDU types, and any associated queues are not used. For example, if we want a connection without reliability, duplicate PDU detection and flow control, no function of the DTCP PM is needed and it can be eliminated along with the associated control PDU types and the transmit and retransmit queues. In that case, the state vector

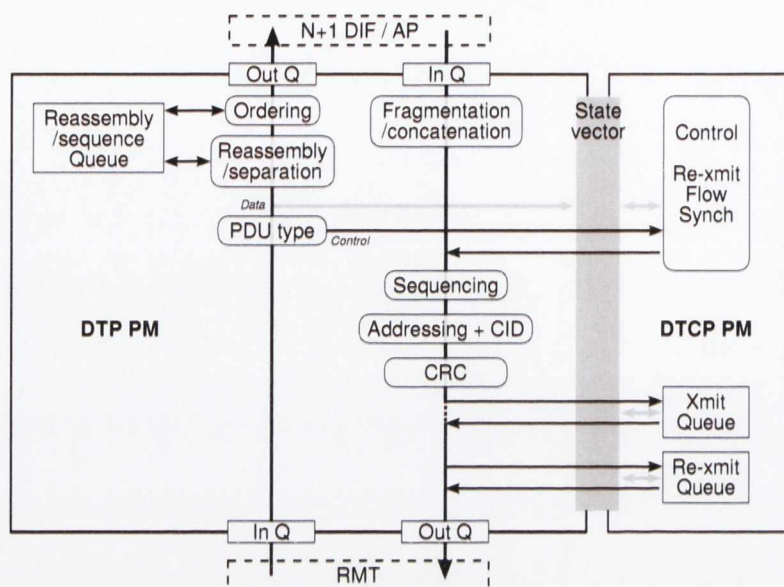


Figure 5.16: The tightly bound mechanisms form the DTP on the left-hand side, while the loosely bound mechanisms form the DTCP on the right-hand side. The two are tied together by a state vector. The figure is based on Day 2008a, Figure 3-5 and Day, Bunch, et al. 2013, EFCP Spec, Figure 1.

can also be eliminated, because only the DTCP PM ever reads from it. The slimmed-down EFCP PM now consists only of the DTP PM and transmits only PDUs of the data type. If some mechanisms of the DTCP are needed but others are not, only the needed parts need be allocated, initialised and used.

The policy-based modular design provides great flexibility and allows a single protocol—EFCP—to span a wide spectrum of connection types and qualities.

Synchronisation

Synchronisation is necessary when the PMs at each end of a connection need to agree on some shared state. Synchronisation information can be explicitly included in PDUs, or it can be derived from information implicit in the exchange of data or control PDUs. In either case, state can be synchronised with the data flow between the two PMs.

An EFCP PM assumes that a PDU that matches its CID can be uniquely identified by its SN. If this assumption does not hold, for example, if SNs roll over before old PDUs die out, or if CIDs are reused for different connections, the PM becomes susceptible to attacks and errors, called ‘hazards’.

Therefore, before data may flow, the two PMs need to make sure that SNs, they will use,

can not overlap with SNs of existing PDUs that may exist in the network for old connections with the same CID.

Fletcher and Watson (1978) and Watson (1981) showed that in order to know when it is safe to reuse a SN for a CID, it is necessary only to estimate the bounds for the sum Δt of three time values: The maximum packet lifetime, MPL , the maximum time between PDU reception and sending an ack, A , and the maximum time a sender will try retransmitting an unacknowledged PDU, R . They showed that if a connection have been idle for $3\Delta t$ or more, all SNs are safe to use. Δt can be bound by conservative values without synchronisation, and can be fine tuned over time. EFCP is based on these insights.

Policies and Mechanisms Many aspects of EFCP's behaviour depend on policies. These policies, applicable to EFCP in, say, layer N , depend on the QoS requirements of the user in layer $(N+1)$, the QoS provided by supporting $(N-1)$ -layers, and layer-wide policies applied across the whole N -layer.

Some aspects, however, are not policies, but are fixed mechanisms that support the variable policies. The separation of mechanisms and policies is central to the design of EFCP.

Here are some examples of variable policies:

1. Data delimiting: Whether or not EFCP exhibits single-packet semantics, stream semantics, or something in-between.
2. Addressing: The format and size of addresses.
3. Reliability: Whether all PDU should reach their destination or not.
4. Ordering: Whether PDU should be delivered in the same order in which they were sent or not.
5. Encryption and compression: Whether data carried should be encrypted and/or compressed or not.

Here are some examples of invariable mechanisms:

1. Support for addresses: A PCI includes addresses for source and destination DIP, except when addresses can be left out entirely (for example due to there only being two DIP in the DIF).
2. Support for sequence numbers: If reliability or ordering is required, sequence numbers are included in the PCI.

3. Integrity: CRC should be included in the PCI to ensure data integrity.
4. Support for fragmentation.

5.2.5.2 The Common Distributed Application Protocol

CDAP is a general-purpose application-independent protocol designed to exchange structured data called objects between applications or perform operations on such objects. CDAP serves a similar purpose as REST in conjunction with HTTP and JavaScript Object Notation (JSON), except that the operations are different from HTTP's, and the serialization format is not restricted to JSON.

Each application will make available its objects in its RIB, which is an abstract object store that can be implemented however the application sees fit (for example in-memory or using a database).

Objects consist of a type, a name, and a value, and can be identified uniquely using the type and name.

CDAP will perform one of six operations on objects: create, delete, read, write, start, and stop. The exact semantics of these operations are defined by the application. The syntax for CDAP is configurable. Current prototype implementations use Google Protocol Buffers (GPB), but any format, including JSON or Abstract Syntax Notation One (ASN.1) is viable.

The DIF use CDAP to exchange management information between the RIB of its DIPs, for example in order to allocate flows, to exchange authentication information, or to implement routing algorithms for updating routing tables across the DIF.

5.2.6 Phases of Operations

For two APs to communicate, the following phases of operations are necessary:

1. Enrolment: The distribution of information and the authentication necessary to enable allocation of flows
2. Flow allocation: The allocation of resources for data transfer
3. Data transfer and synchronisation: The actual data transfer and management of connection state

5.2.6.1 Enrolment

Enrolment is the distribution of necessary information and the authentication of relevant entities to allow IPC to be initiated.

Abstractly, enrolment can be separated into *registration* and *activation*. Registration is the exchange of data so that adequate information is available at the relevant locations to allocate flows and the readiness of the network to provide the resources for such flow instances. Activation is the granting of permission to actually create said flows. The separation is useful so that systems' 'offline' or 'online' status can be toggled with activation/de-activation without going through the more elaborate process of registration/de-registration each time. A system that is registered but not activated cannot create new flows, but existing flows persist. De-registration removes the information necessary to create new flows. Existing flows may survive until they need registration information, the lack of which might provoke the abortion of the flows.

Concretely, in RINA, enrolment of a system either means one of its DIPs joining an existing DIF; or the creation of a new DIF from one of the system's DIPs that other systems' DIPs may subsequently join.

5.2.6.2 Allocation

Allocation is the allocation of resources for creating and maintaining flows, and the instantiation of flows.

In RINA, a flow and a connection are not the same: A connections are between EFCP PMs, while flows are between AEs.

In other architectures, and specifically in TCP/IP, they are conflated.

5.2.6.3 Data Transfer and Synchronisation

This phase involves the transfer of the actual data and manages the shared state (if any) of a connection.

5.3 Discussion

RINA is designed using a black box approach: It exposes a well-defined service, IPC, via the RINA API. The stated purpose of the RINA architecture is to provide this service as completely and simply as possible (Day 2008a; Day, Bunch, et al. 2013).

5.3.1 Properties

Below, we discuss our properties as they relate to RINA.

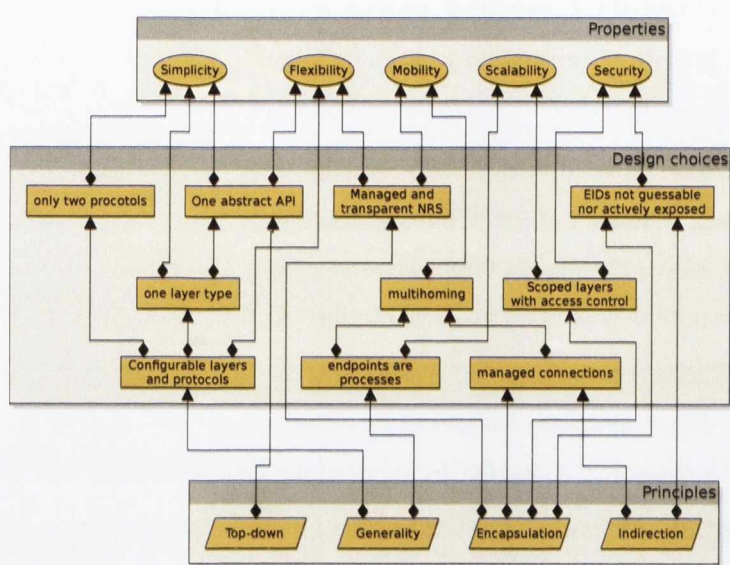


Figure 5.17: Properties and principles in RINA

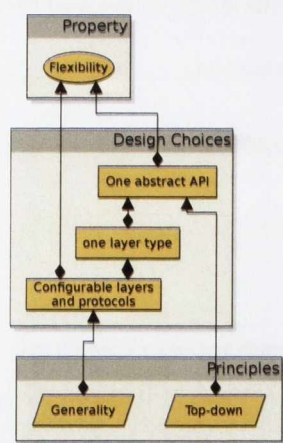


Figure 5.18: Flexibility property in RINA

5.3.1.1 Flexibility

Configurable Layers and Protocols Each layer is configurable: Name resolution, Routing algorithms, Addressing, Encryption, and PDU formats and encoding are customisable, and might depend on, for example, network technologies or scope. Once instantiated, the QoS of individual flows in the layer are configurable at run time by the application. QoS parameters include jitter, throughput, reliability or delay.

Single Well-defined API Layers (aka. DIFs) in RINA have a separation of API, mechanism, and policy. The main API is invariable and implementation-independent and is summarised in Section 5.2.1. Because a layer is a special case of a distributed application, the API between layers is the same as that between layers and applications. This allows flexibility in how layers are composed, as well as in which layer an application interacts with—the API for accessing a low-level layer used only for local LAN or link connectivity is the same as a high-level Internet or VPN layer.

Managed Name Resolution Name resolution is handled by and encapsulated by the layer, so any change to name resolution affects only one layer and not any applications.

5.3.1.2 Mobility

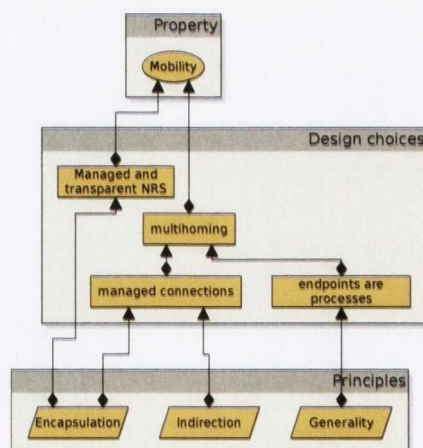


Figure 5.19: Mobility Property in RINA

Managed Name Resolution The RINA API uses AE names to identify IPC endpoints. The names are independent of network location and addressing, and RINA handles name resolution without involvement from applications, which then do not depend on NRS APIs,

mechanisms, or data formats. Address changes can be handled transparently to applications, which improves mobility.

Managed Connections RINA manages connections. Application flows can be multiplexed across several connections, which may be created or terminated dynamically dependent on policies and network environment changes. Broken connections and recovery is handled by the layer. This provides for stability and multihoming, and enables mobility.

5.3.1.3 Scalability

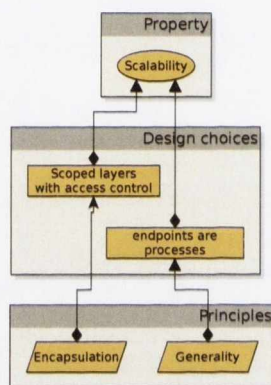


Figure 5.20: Scalability Property in RINA.

Scoped Layers All layers are scoped. Like the Internet, lower layers usually have a smaller scope than upper layers. Routing, addressing and name resolution can be tailored to each layer based on for example scope, as well as other characteristics such as network technologies, expected network dynamism, etc.

Scoping in this context also means isolation: All routing information is local to the layer. With planned scoping, dependencies on routing information from irrelevant parts of the network is eliminated. This locality of routing data can greatly improve scaling.

A few layers will still need to have global span. In such cases, upper layers' addressing and routing can be coordinated those of lower layers to improve scalability, as proposed by Day, Trouva, et al. (2011).

Routing Entities Are Route Endpoints Day points out that IP addresses identify a point on the path to the destination: The destination is always a process (whether or not a user or an OS process) and it is available via one or more IP addresses. Routing on the identifier of

a point of the path to the route's destination will lead to routing tables that are at best the same size as routing tables, where the real endpoint is used as routing identifiers.

In Section 4.4.1.3, we show that there is a factor four difference between number of PoA and number of nodes on the Internet. RINA routes on the nodes, which leads to smaller routing tables than if routing was done on PoAs like in TCP/IP.

5.3.1.4 Security

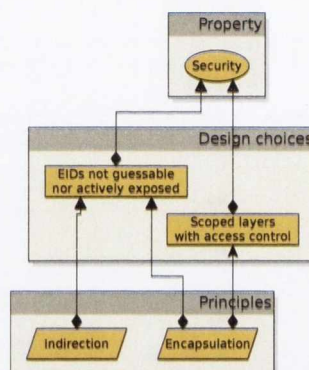


Figure 5.21: Security Property in RINA

Scoped Layers The scope of layers (aka. DIFs) is enforced both by physical access limitations and by access control. The latter means that a node (i.e. a DIP) might need to authenticate before it can join a layer, if the layer so requires. If a node has not joined a layer, it cannot send or receive data using that layer. Layers can quickly lock out and isolate nodes that turn out to be hostile or faulty. This greatly reduces attack vectors and DoS possibilities. This functionality is independent of applications.

It is worth noticing two things about the authentication of DIPs in order for them to join a DIF:

1. The authentication and authorisation policies are configured for each DIF and may vary. Indeed, a DIF may skip authentication all-together and be publicly available (i.e., it can be joined by any DIP that so wishes).
2. The authorisation and authentication of a DIP is managed by RINA, based on configurations and policies. It is different from authentication and authorisation of APs using that DIF. The latter happens between user-applications, and a prerequisite to even exchange such application authorisation messages is for the DIPs to which the APs between which authorisation needs to take place are already part of the same DIF,

i.e., that they are themselves authorised—otherwise, no messages can be exchanged between them at all.

Additionally, because addressing and name resolution is layer-internal, addresses are not obtainable or guessable for unauthenticated adversaries.

Non-guessable EIDs The EIDs contained in PDUs are assigned independently of addresses and port numbers. Their only property is that they are unique within the DIP of the endpoint; they reveal no other semantics. Therefore, they cannot be guessed from inferential information or conventions. However, they can still be intercepted from existing traffic flows. Nevertheless, without being able to guess EID, attackers cannot focus their attack.

Configurable Layers and Protocols A layer can be configured to encrypt its traffic. Key exchange mechanisms and encryption methods are a matter of configuration policy, as is which part of the PDU gets encrypted. All this is transparent to applications. They will still need to apply their own encryption, if they want to keep their traffic secret from the layers they interface with.

5.3.1.5 Simplicity

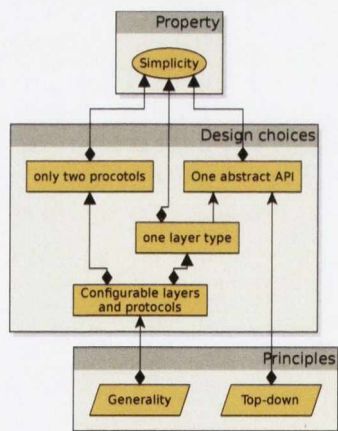


Figure 5.22: Simplicity Property in RINA

One API In RINA, a layer is a specialised distributed application that provides IPC. Because a layer is an application, the API between a layer and an application is the same as that between layers. This has two consequences: First, applications are not special-tailored to any one layer but can utilise whichever layer has the requisite QoS and scope. Second,

layers can be arbitrarily combined without special-tailored layer integration. The effect is that any specialisation and custom-made integration is eliminated from both layers and applications, simplifying the architecture.

One Layer Type In RINA, instances of the same layer is repeated with different configurations to build the IPC infrastructure. The effect is that there is only one general layer to implement, which can be customised to different uses, rather than several specialised layers, each of which can only be used in a special setting.

Minimum Number of Protocols There are only two protocols in RINA: EFCP for transport and CDAP for applications. The generalisation of special-purpose transport protocols into the configurable EFCP, encompassing behaviour today covered in MPLS, Ethernet, UDP, TCP, the Real-time Transport Protocol (RTP), Transport Layer Security (TLS), etc., reduces complexity.

CDAP can be used to create higher-level application protocols, just like HTTP can. An example of which is any REST API. Whereas HTTP depends on various other elements of the TCP/IP architecture, such as DNS names, IP addresses, and TCP ports, CDAP has no dependencies on EFCP and can run on any transport protocol, including TCP.

5.4 Principles

RINA is built on the following general principles:

1. Encapsulation
2. Indirection
3. Generalization
4. Recursion
5. Simplicity

The RINA literature is explicit about this being the case. For example, Item 2 is mentioned as a necessity in network addressing (Day 2008a, p. 166), Item 3 is exemplified in Day 2008a as the separation of mechanism and policy, and Item 4 of course has a prominent role. The goal of Item 5 comes from Newton's Principia (Day 2008a, p. 58), and is a goal for natural sciences and maths alike, and therefore also for computer science, which have a leg in both camps.

Figure 5.23 shows the listed principles and our view of their relation to the goal of simplicity. Encapsulation and indirection help to lower the coupling between components, which supports modularity. Locality increases the cohesion of components, also supporting

modularity. Modularity, generalisation, and recursion are all principles that can help simplify systems.

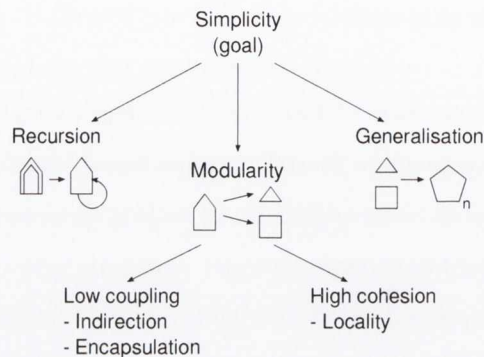


Figure 5.23: Principles in RINA

In the following, we give examples of where these principles are applied in RINA and why.

5.4.1 Encapsulation

With ‘encapsulation’ we mean data and function hiding. The point of encapsulation is to minimize the coupling between components by minimizing what they know about each other. Terms such as ‘black box’ or ‘layer’ are used for components that expose to the outside only what is strictly necessary and encapsulate everything else.

5.4.1.1 Addresses

Addresses are DIF-internal, which means addresses are not required by nor exposed to any component outside of the DIF. Processes outside of a DIF will therefore generally not know the addresses to use to construct PDUs to reach a specific DIP within that DIF.

If the DIF does not encrypt its own traffic, however, the addresses may still be intercepted by adversaries in lower layers.

Because no outside components are expected to know or care about the addresses used within the DIF, the DIF can choose an addressing and routing scheme to match its needs—as long as its DIPs can comply. This allows a DIF to, for example, choose a small address space for small networks, reducing the size of the address fields in PDUs, potentially increasing the ratio of payload bytes (SDU) to non-payload bytes (PCI) for transmitted PDUs leading to higher effective throughput; or to choose a structure for the address space particularly suited for efficient routing in the specific network the DIF covers.

Addressing and routing is thus a DIF-local decision, and not a global one as is the case with the Internet.

5.4.1.2 Connection and Endpoint Identifiers

An EID is internal to the DIP in which it is created. The CID is composed of two EIDs and is shared among the two communicating DIPs. If the EID identifier space is chosen to be large enough, and the EIDs are assigned in an unpredictable way, the CID cannot, in practice, be guessed. This eliminates attack vectors based on known ports and port scanings as known from the Internet. Also, the fact that the communicating DIPs are able to generate new CIDs based on local information only without the need to consider other components means that new connections can be created effortlessly and in abundance. This helps support a flow when multiple connections are desirable or when old connections can not be used any more, for example, when endpoints move or when a connection runs out of sequence numbers.

5.4.2 Indirection

Indirection is when one identifier is mapped to another identifier from the same or another name space. Indirections can be used to decouple components by letting each use their own internal identifiers for shared entities where common identifiers would otherwise have been used. In other words, indirection enables encapsulation and locality.

A good example of indirection is the now ubiquitous use of virtual memory in operating systems: Programs know about and operate in a virtual address-space, while the OS maps it to a physical address-space, closely tied to the hardware of the system. This hides changes in hardware from the programs, and allows the OS to manage memory, improving flexibility and resource management of the system. The cost in performance by maintaining the indirection needs to be considered and compared to the added flexibility.

Application names (ANs) are the only global identifiers used in RINA. They are used in the API between DIFs and AEs and in-between DIFs to identify the source and destination of a flow.

The ANs are via the API's *allocate* operation bound to a FAI in the DIF responsible for managing the connections of the flow. The application is provided with a PID, which is a handle that enables further operations on the flow. The mapping of application names to addresses happen each time a new connection needs to be created for the flow. This allows for address changes, for example when applications move in the network, which enables mobility.

5.4.3 Locality

With 'locality' we mean that information needed to perform some function is available locally. For example, if routing information is continually propagated to all nodes in a network so that each node can make forwarding decisions based only on information from its local routing table, routing decisions can be made much faster than if each node has to consult a remote database or a set of distributed tables each time. Locality provides independence from other components and can eliminate inter-component communication in time-critical situations, which decreases delays and increases performance.

5.4.4 Generalisation

Generalisation is when a set of different concepts can be described with a common concept.

An example of various levels of generalisation is the relation between super-classes and sub-classes in object-oriented programming: A super-class is a generalisation of one or more sub-classes.

Generalising reduces the number of concepts or components. This in turn leads to fewer interfaces and thus a simpler architecture.

In RINA, there are two areas where we find examples of generalisations: Layers and Protocols.

5.4.4.1 Layers

In RINA, all layers are specialisations of a generalised layer that provides one specific service: IPC. Layer instances have the same general mechanisms, but they vary in scope and in policies. Policies provide for a range of available layer characteristics, such as routing schemes, data and address formats, and protocol behaviour, so that the same general layer can be used for layer instances providing IPC over many network types including point-to-point wires, small-scale LANs, global internets, VPNs, virtual networks, etc.

This contrasts the way layers in the Internet model are defined. Modeled after the layer concept as introduced in the THE OS Dijkstra 1968, where layers provided different services, each layer in the Internet is also thought to provide different services using different (but sometimes similar) mechanisms. Despite overlaps in functionality, especially between L2 and L3, no generalisation is attempted.

The generalised layer in RINA makes for a simpler architecture that require fewer concepts to understand, build, use, and improve a solution. The specialised layers in the Internet model have the opposite effects: They make it more complex, there are more details

to understand that cannot be reused, it is harder to implement, and it has proven almost impossible to change (Floyd 2002).

5.4.4.2 Protocols

In RINA, there are two core protocols: EFCP and CDAP.

CDAP provides the building blocks to create general higher-level distributed application protocols using an object-oriented abstraction model. CDAP is also used for network management and routing information updates within RINA itself. CDAP's primitive operations includes create, delete, read, write, start, and stop, and work on an object name-space independent of any underlying transport protocol. The scope of the object names is limited to one distributed application. CDAP could be used to implement various distributed applications that currently use special-tailored application protocols, for example the Web, the Secure Shell Protocol (SSH), the File Transfer Protocol (FTP), or email.

EFCP is a general transfer protocol that provides the same service in one protocol as the combined services provided by the Internet's transport layer protocols (TCP, UDP, SCTP, etc.).

In RINA, the generalisations of protocols, both on the transport-level and application-level, reduces the number of protocols and associated implementations in operating systems and in applications drastically from several hundred to just two. Via adjustable QoS parameters, EFCP can be made to fit a large range of network types supporting many different services, and by combining the primitive operations listed above, CDAP can be used to construct any object-oriented application protocol.

5.4.5 Recursion

Recursion is when something is defined in terms of itself. Recursion can simplify an architecture because the number of components and services (or interfaces) can be reduced when components can be defined in terms of themselves. As an example, let us look at the recursive layers in RINA and the non-recursive layers in the Internet. Let us describe the layer f , which depends on the service A in order to provide the service B , by the morphism $f : A \rightarrow B$.

In both cases, an IPC-service is provided. In the Internet, the service is provided via the socket API, S , while RINA provides it via the RINA API, R . Let us describe Internet applications a_i as morphisms of the type $a_i : S \rightarrow X$, and RINA applications a_r as morphisms of the type $a_r : R \rightarrow X$, where we think of X as an unspecified service provided by the

applications. Both a_i and a_r needs to be able to run on the physical network, which we denote as the pseudo-service P .

In the Internet, S is supported by the transport service T . The OS o provides the morphism $o : T \rightarrow S$. The transport service T is provided by the transport layer t and depends on the IP-based Internet service I : $t : I \rightarrow T$. The network layer n requires the Ethernet service E , so it provides the morphism $n : E \rightarrow I$. E is provided by the link layer l , which runs on the physical network: $l : P \rightarrow E$. How can we create a combined Internet service $i : P \rightarrow S$ running on the physical network that can support the application a_i which depends on S ? The answer is to compose the layers: $i = o \circ t \circ n \circ l : P \rightarrow E \rightarrow I \rightarrow T \rightarrow S$. The resulting architecture has eight components: Four services (in addition to the physical network pseudo-service) and four morphisms; and they are all fixed in relation to each other, yielding one possible configuration.

In RINA, each DIF d is a layer that provides the RINA service R in terms of another lower-level RINA service: $d : R \rightarrow R$. At some point, there is a degenerate bottom layer that provides the RINA service based on the physical network only: $d_0 : P \rightarrow R$. In order to create a RINA service $r : P \rightarrow R$ to support a RINA application $a_r : R \rightarrow X$, one has to combine the bottom layer with any number of RINA layers: $r = d \circ \dots \circ d \circ d_0 : P \rightarrow R \rightarrow \dots \rightarrow R$. The resulting architecture has three components: One service in addition to the physical one, and two morphisms. Because of the common service R , any number of d morphisms can be arranged in various orders, yielding an very flexible configuration-space.

In conclusion: While the Internet requires eight components for a fixed configuration, RINA requires only three components for a flexible configuration. Recursion has simplified the architecture by reducing the number of different components, while providing additional flexibility.

5.5 Summary

RINA is designed with all four taxonomy principles in mind. Indirection is used in several places: Between ANs, which are used as FEIDs identifying each flow, and CEIDs, identifying the connections, so that multihoming can be supported by associating several connections with one flow; between CEIDs and addresses, so that changing addresses do not break connections; and between forwarding addresses and ports, similar to in IP routing tables, so that the network graph may change without address renumbering.

Indirections ensure that only flows, identified via ANs, are exposed through the APIs.

This encapsulates addresses, CEIDs, name resolution, and connections inside the layer. Address formats and routing algorithms can therefore vary inside the layer, without affecting the API and the applications using the layer. The CEID format can also be customised per layer. Name resolution, i.e., the translation from ANs to addresses, is also encapsulated per layer, and can therefore be customised and changed at the layer owner's discretion without affecting applications. Connection management including error recovery when connections break is handled inside the layer, allowing the layer to guarantee a certain QoS because it knows how errors are handled.

In addition, encryption and compression may be employed by the layer, protecting and/or compressing its PDUs before they are handed over to lower layers.

The APIs are designed in an extensional way—they are kept simple and depend on high-level RINA concepts only. They do not depend on or expose protocol semantics, syntax or other implementation details, so these may change without it affecting the APIs and the users of the APIs. This is what makes the architecture flexible and adaptable to future changes.

The extensionality of the service and the APIs, the encapsulation of functions and identifiers, and RINA's general focus on separating mechanisms and policy makes the architecture configurable and general.

Table 5.2 shows the taxonomy score for RINA. As we have argued above, RINA employs all four principles and we therefore assign the maximum score of 2 for each.

As we have also argued, this leads to RINA obtaining all five properties as well: It is a flexible architecture that allows for changing implementations; it provides excellent mobility; its multiple and adaptable layering provides scalability; security is provided per-layer as well as per-flow; and the recursiveness, generality and configurability of the architecture means that only one layer needs to be implemented in order to provide a full multi-layer architecture making the architecture simple. However, the one layer that needs to be implemented is complex because it needs to be generic and highly configurable for it to be able to cater for the wide array of usages expected, ranging from VPNs at high layers over something resembling a combination of today's transport and network layers in the middle layers, to high throughput back-bone specialised tunnels at low layers. This complexity prevents an overall top score.

IA	Principles				Properties				
	Ext	Gen	Enc	Ind	Flx	Mob	Scl	Sec	Sim
RINA	2	2	2	2	2	2	2	2	1

Table 5.2: Principles and Properties Scores for RINA

Chapter 6

NDN

We here present NDN (Zhang, Afanasyev, et al. 2014), which is the continuation of CCN (Jacobson et al. 2009). It defines itself as a network architecture that provides a universal data distribution service. It is one of many such architectures that collectively are termed Information Centric Networking (ICN). Examples of other ICN architectures are DONA (Koponen, Chawla, et al. 2007) and Named Information (NI) (Dannewitz et al. 2013).

We argue that NDN is inherently non-extensional and that it does not encapsulate much functionality. It employs almost no indirections. This leads to a non-general, inflexible, non-scalable architecture, with limited support for mobility. It focuses on security, and is overall quite simple, at the cost of pushing most functions to the applications.

6.1 Premise

CCN (Jacobson et al. 2009) is based on the premise that many of the problems of the Internet is caused by a mismatch between the service provided by the Internet, based on *where*, and the abstractions used by applications, based on *what*. CCN equates *where* with hosts and *what* with content. A new network service based on named content, manifested in CCN, closes the gap, and the hypothesis is that this should solve the scalability, security, and mobility problems of the Internet and enable new solutions.

NDN (Zhang, Estrin, et al. 2010) is the continuation of CCN and is built on the same premise. In Zhang, Afanasyev, et al. 2014, the semantics of names, which in Jacobson et al. 2009; Zhang, Estrin, et al. 2010 identified only content, is expanded to identify everything: In Zhang, Afanasyev, et al. 2014 endpoints, data chunks, and commands are given as examples.

6.2 Model

NDN is built bottom-up with its protocols as its primary invariants.

6.2.1 NDN According To NDN

Jacobson et al. 2009; Zhang, Afanasyev, et al. 2014; Zhang, Estrin, et al. 2010 place the NDN architecture in the network layer and define it as a ‘universal overlay’ providing a content distribution service.

The architecture was originally built around a content-centric concept meaning that each PDU contains a name used to identify a data object rather than an addresses used to identify source and destination endpoints (Jacobson et al. 2009; Zhang, Estrin, et al. 2010). This is later generalised so that the name in a PDU ‘can name anything—an endpoint, a data chunk in a movie or a book, a command to turn on some lights, etc.’ (Zhang, Afanasyev, et al. 2014, Section 1). For brevity, we usually refer to the entity identified as *data*.

The authors of NDN argue that the resulting ‘distribution network’ provides a more general service that is better suited for the way applications should utilize a global network than the ‘communication network’ service that the Internet provides.

NDN is centred around two types of PDUs: Interests and data; two types of applications: Consumers and producers; and one kind of in-network nodes: NDN forwarders. A consumer obtains data by first sending out an interest PDU whose name identifies the piece of data the consumer wish to obtain. The NDN forwarders in the network are responsible for forwarding the interest PDU to a producer capable of producing the data. The producer then creates a data PDU, which is sent back along the same path in the network as that of the interest that triggered the data production. The data PDU may be cached in the forwarders on the path from the producer to the consumer so that subsequent interests for the same data can be satisfied by a forwarder holding a cached copy. The authors argue that such ubiquitous caching can minimise repeated transfers of the same pieces of data across the same network segment, reducing overall network traffic (Zhang, Estrin, et al. 2010).

An NDN forwarder is shown in Figure 6.1. PDUs from and to other forwarders as well as applications enter and leave via the faces. Decisions as to which faces interest PDUs are forwarded to are based on the information contained in the forwarding information base (FIB). How the FIB is kept up to date with routing information is only suggested in Jacobson et al. 2009 and later concretised in Hoque et al. 2013; Wang et al. 2012, where the routing scheme is built on OSPF used for IP.

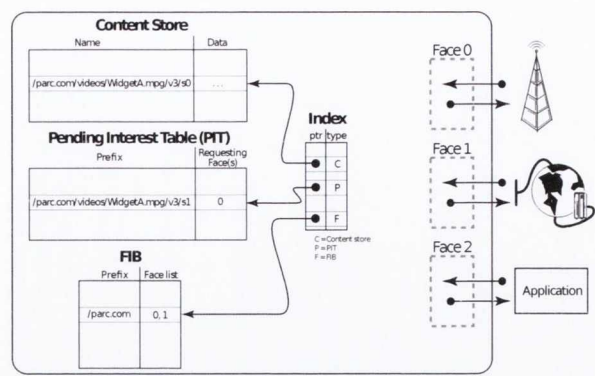


Figure 6.1: An NDN Forwarder. Figure courtesy of Jacobson et al. 2009; Zhang, Estrin, et al. 2010.

Names in NDN are hierarchical and consist of a series of name components. Forwarding decisions of interest PDUs are based on longest-prefix matching of the name in the PDU with the names contained in the forwarder’s Forward Information Base—the so-called FIB. This allows aggregation so that one FIB entry containing a name prefix can be used for forwarding all interests with names matching it. Longest-prefix means that if there are multiple matching entries in a FIB then the entry with the longest matching prefix will be used. This is the same mechanism used for forwarding in IP.

The forwarder’s pending interest table (PIT) provides a ‘breadcrumb’ to route data PDUs back towards the origin of its associated interest. When an interest reaches a forwarder, the PIT is updated so that matching data can be sent back in the direction from where the interest originated. If a matching PIT entry already exists when an interest arrives, the PIT entry is updated but the interest is dropped. When a data PDU is forwarded through the forwarder, the PIT entry is cleared and the data PDU is forwarded to all outputs specified in the PIT entry. Any data PDU that does not match a PIT entry are dropped. This forwarding scheme de-duplicates interests and data along the same path but maintains necessary duplicates for different paths.

6.2.2 The NDN Architecture

The NDN architecture is built around the two primitive PDU types: interest and data, and two types of APs namely producers and consumers. Consumers send interest PDUs on to the network. Each interest contains the name of a piece of data that can fit into a data PDU. The network is expected to route the interest PDUs to a producer capable of providing data PDU with data matching the name specified in the interest.

NDN retains the layer model from TCP/IP, and describes itself as an evolution of the ‘universal network layer’ L3, that instead of providing a service that delivers a packet to a given destination address, it provides a service that fetches data identified by a given name (Zhang, Afanasyev, et al. 2014, Section 1).

6.3 Discussion

In this section, we discuss our selected properties in the light of NDN.

6.3.1 Properties

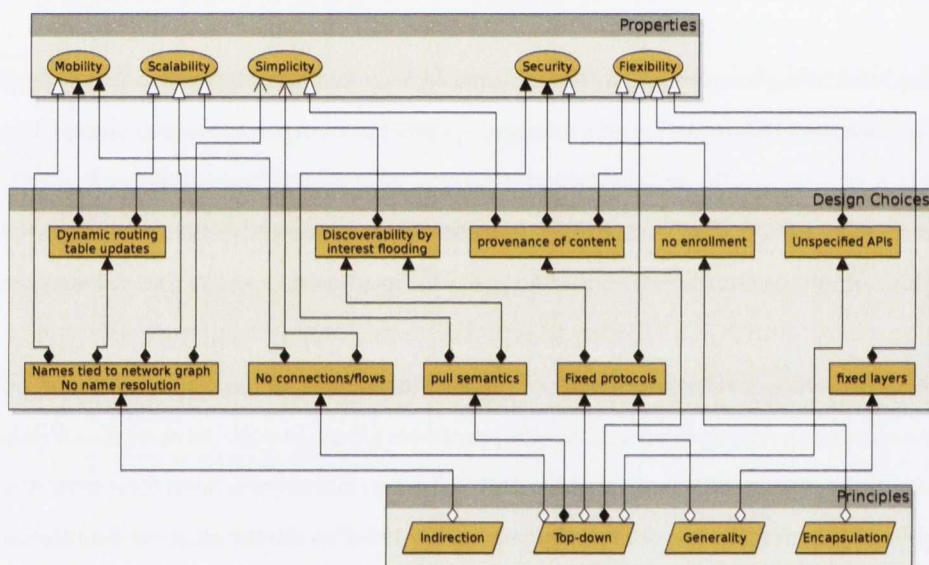


Figure 6.2: NDN Principles and Properties

6.3.1.1 Flexibility

Unspecified APIs There are no abstract APIs defined for NDN: No list of allowed operations and their parameters, no description of actions that applications should be able to invoke, no specification of how the NDN service should be used. The result is fragmentation and several APIs have emerged: For example, each of the NDN implementations CCNx (CCNx 2016) and NDN-CCL (NDN-CCL 2016) have their own API, both of which depends on the particular implementation and particular version of the NDN protocol that each uses.

Both NDN APIs are low-level: They provide libraries for an application to build, interpret, send and receive individual NDN PDUs. According to Tanenbaum and Wetherall 2011, this

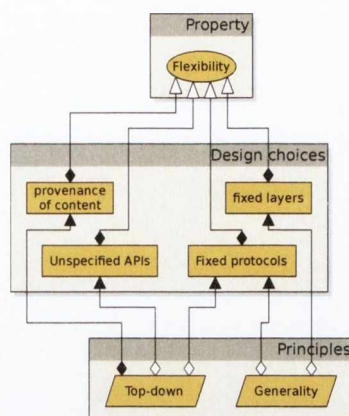


Figure 6.3: NDN Flexibility Property

is the epitome of bad network design.¹

Provenance NDN requires provenance: Data is bound to a unique name by a signature by their publisher. In NDN, this is done per PDU: The publisher needs to fragment its data according to allowable PDU sizes, and manage the granularity of names accordingly. This, along with an initial lack of high-level APIs, might have led to the coupling of PDU construction and data-generation seen in CCNx and NDN-CCL, where the application itself constructs and signs its PDUs and passes the whole PDU to the network. Any change in PDU format or signing mechanism that is not already parameterised may require changes to applications.

Fixed Protocols With Fixed QoS The NDN protocol does not provide transport services, such as reliability, ordering, retransmission, or (de)multiplexing. As such the QoS of NDN is fixed, and applications do not have the flexibility to choose QoS parameters. Instead, any QoS not provided by NDN has to be implemented by the application.

Fixed Layers The layer model in NDN is built on that of TCP/IP. NDN places itself in L3, running on L2, for example on Ethernet, or in between L3 and L4, running on L3, for example on IP. However, NDN applications interact with NDN forwarders by exchanging lower-level PDUs with it, for example using UDP or TCP. This locks applications and forwarders to the

¹Tanenbaum and Wetherall 2011, Section 1.3.5 emphasises the distinction between the protocols and the service of a layer. According to the authors, ‘a service is a set of primitives (operations) that a layer provides to the layer above it’. They also write: ‘Many older protocols did not distinguish the service from the protocol. In effect, a typical layer might have had a service primitive SEND PACKET with the user providing a pointer to a fully assembled packet. This arrangement meant that all changes to the protocol were immediately visible to the users. Most network designers now regard such a design as a serious blunder.’ (our emphasis).

specific lower layer.

6.3.1.2 Mobility

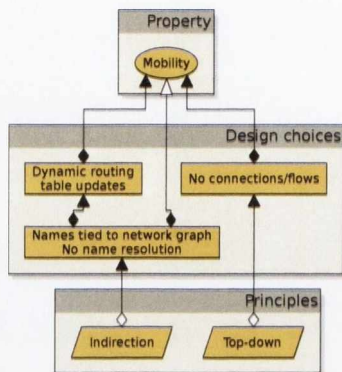


Figure 6.4: NDN Mobility Property

Names Used In Routing NDN names are used directly in routing. This makes the names tightly coupled with network routing and the network graph, despite incorrect claims that names are opaque to and independent of the network.²

The lack of indirection between names and routing identifiers (aka. ‘addresses’)—provided in TCP/IP by DNS, and in RINA by each layer’s RIB—makes mobility hard to supported: Every time a producer or piece of content changes location, the network routing tables that are used to route interest PDUs have to change too.

Dynamic Routing Table Updates While interest PDUs are routed based on long-lived routing table entries, data PDUs are routed based on the temporary routing state induced by interest forwarding (i.e., the interest ‘breadcrumbs’). This state is short-lived, and changes for each interest. If the state is invalidated before a data PDU has time to travel back along the path of the corresponding interest, a timeout is meant to ensure that a new interest will be sent out. While the timeout mechanism and interest retransmission is left to the application, when done correctly, it does provide consumer mobility.

To support producer mobility, the long-lived routing tables used for interest routing need to be updated whenever the producer moves. If the producer’s new location breaks

²Zhang, Afanasyev, et al. 2014, Section 2.1: ‘Although routers recognize boundaries between components in a name, they attribute no meaning to names, i.e., NDN names are opaque to the network. This design decision allows each application to choose the naming scheme that fits its needs, and naming can thus evolve independently from the network.’

aggregation, this requires global routing updates. Proxy names, analogously to MIP, may be used to prevent aggregation from breaking.

No Connections NDN does not have connections because it does not provide a transport service requiring connections. Any mobility problems caused by invalidated connections, due to for example network graph changes during the lifetime of a connection, is non-existent in NDN.

However, connections are still needed for all applications that require transport services (e.g., reliable data transfer)—only in NDN, these functions will have to be implemented by applications because the architecture does not provide them.

6.3.1.3 Scalability

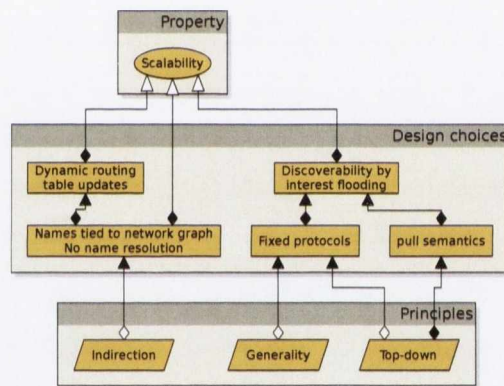


Figure 6.5: NDN Scalability Property

Names Used In Routing Because names are used directly in routing tables, scalability depends on whether or not names can be aggregated in routing tables. This depends on whether or not names correlate to the network graph. If names are to be independent of location, as NDN advocates, names will not in general correlate with the network graph.

Secure Namespace Mapping (SNAMP) (Afanasyev, Yi, et al. 2015) introduces a loc/id solution, similar to what has been proposed for TCP/IP with, e.g., LISP (Farinacci et al. 2013). This has scalability problems too, as described in Day 2008b; David Meyer and Lewis 2009.

Dynamic Routing Table Updates Producer mobility facilitated by updating routing tables every time a producer moves does not scale except if names are designed to prevent this by

being bound to their network locations to allow aggregation, and mobility is restricted, so that any new location still allows for aggregation, thus keeping routing changes local.

6.3.1.4 Security

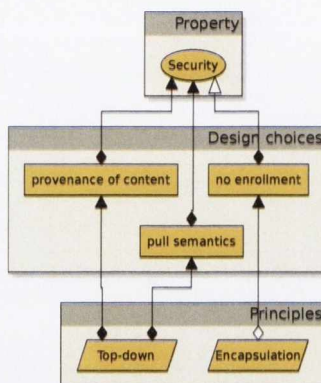


Figure 6.6: NDN Security Property

Pull Semantics The pull-semantics of NDN means that only interest PDUs can be used to target servers. Data PDUs cannot target specific victims because they have no destination address, but instead rely on routing state induced by a preceding interest PDU. Interest PDUs can not contain arbitrary data and do not require the same amount of processing as data PDUs.

This eliminates attack vectors that rely on large data PDUs being sent to overflow the victim’s processing capabilities or bandwidth, or on manipulating destination addresses of response PDUs, which includes amplification attacks.

Attacks using interest PDUs are still possible, as documented in (Afanasyev, Mahadevan, et al. 2013), but can be mitigated.

Provenance NDN integrates provenance. From a security perspective, this is both good and bad: When there is never any doubt as to who has published a certain piece of data, it is straightforward to assess whether or not to trust it, but at the same time, it is harder to anonymise data.

No Scope or Enrolment Like in TCP/IP, there is no authenticated enrolment in NDN. This means that anyone can join the network and send interests. Lack of scoping means that any producer is reachable from everywhere in the network, if the names it serves are known.

6.3.1.5 Simplicity

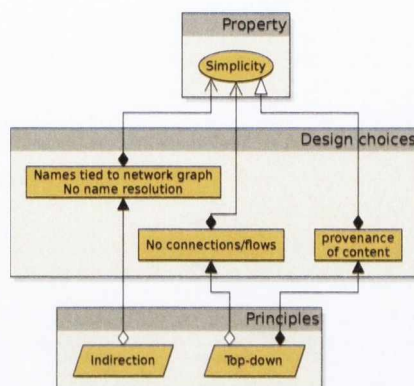


Figure 6.7: NDN Simplicity Properties

Names Used In Routing Names are used in routing and there is no NRS. This makes the architecture simpler. On the other hand, if the indirection that would otherwise be provided by an NRS is needed, e.g., for mobility purposes, applications will have to implement that themselves, pushing complexity to the applications. SNAMP (Afanasyev, Yi, et al. 2015) was designed to provide loc/id, but requires additional application support and therefore additional complexity.

NDN, according to Zhang, Afanasyev, et al. 2014, eliminates address management. However, because names are used for routing, names have to be managed to allow for aggregation and scalability. This is the responsibility of applications, which will then need intricate knowledge of the interaction between the network graph and name space. This greatly increases application complexity, and regardless, applications might not have the adequate knowledge to do this effectively.

No Connections There are no connections in NDN and no transport services. Any application that requires a transport mechanisms, such as reliability, needs to implement this itself. This reduces complexity of the architecture, but pushes complexity to the applications.

Provenance NDN requires that publishers sign all PDUs so that they can be validated by NDN or by consumers. This requires that the applications that sign the PDUs do it in a way compatible with how NDN validates it. This introduces a dependency between applications and NDN, increasing complexity.

6.3.2 Bad APIs

Tanenbaum and Wetherall 2011, Section 1.3.5 emphasise the distinction between the protocols and the service of a layer. According to the authors, ‘a service is a set of primitives (operations) that a layer provides to the layer above it’. They also write: ‘Many older protocols did not distinguish the service from the protocol. In effect, a typical layer might have had a service primitive SEND PACKET with the user providing a pointer to a fully assembled packet. This arrangement meant that all changes to the protocol were immediately visible to the users. *Most network designers now regard such a design as a serious blunder.*’ (our emphasis).

The APIs of CCN (CCNx, CCNx 2016) and NDN (NDN-CCL, NDN-CCL 2016) are exactly that: A set of operations that allows sending, receiving, building, and parsing CCN PDU (i.e., interest and data PDUs). It is poor API design, because it is not abstract at all, but very tightly tied to the protocols in terms of syntax, formats, and semantics.

Similar to TCP/IP, we argue that this is caused by a protocols-first viewpoint behind the architecture. Instead of first specifying a (black box) service with the required properties followed by a protocol design that supports that service, the CCN/NDN community has done the opposite: The main protocol has been thoroughly specified in terms of forwarding mechanisms and types of PDUs, but nowhere is the service this protocol provides specified concretely, for example, as a set of abstract operations. The closest is the API documentation for CCNx and NDN-CCL.

When protocols become the central anchor of the architecture, they at the same time become the invariant that everything else leans on, including the APIs. If it turns out later that some aspect of the protocol design needs to change, for example, naming or routing, everything built on the premise that those were invariant becomes obsolete. The rigidity that we have seen in TCP/IP prevails; in fact, in NDN, it is worse because the socket API does provide a level of abstraction (e.g., for TCP, a stream abstraction) and does not require applications to build their own PDU. Also, TCP/IP manage transport mechanisms such as sequencing and retransmissions; in NDN this is left for applications.

6.3.3 Addresses, Names, and Indirections

We have described how routing in IP relies on aggregation for scalability and how aggregation relies on the address-space to be aligned with the network graph. In other words, the premise for aggregation is that if addresses are ‘near’ to each other in the address space, they are ‘near’ to each other in the network (for some sensible definition of nearness). So, there should be a close relation between addresses and the network graph.

NDN routing relies on the same technique and has the same requirement. However, routing in NDN is done on content names directly. Content is explicitly *not* tied to the network location; in fact, the whole point of NDN is to create a location-independent abstraction working in terms of ‘what’ (content and content names)—not ‘where’ (locations, hosts and IPs), to extract the central ideas of Jacobson et al. 2009, Section 1.

This fundamental contradiction has been neglected in NDN until very recently with the introduction of SNAMP (Afanasyev, Yi, et al. 2015): A mapping mechanism for NDN between names used for routing and names used as identifiers. This is a concept similar to loc/id that is the foundation for LISP, HIP, and ILNP.

In essence, SNAMP introduces a required indirection between different kinds of identifiers, without which routing is not scalable. However, the loc/id concept it builds on is problematic as discussed in (Day 2008b; David Meyer and Lewis 2009).

Furthermore, a tempting question is: Why should routing be done on NDN names in the first place? The point seemed to be to eliminate the indirection provided by DNS; now another indirection has been introduced in its place.

The focus on routing and protocols have clouded the view from what is the essential question: What service should NDN provide, what should its API look like, and how should a protocol be designed that best supports it? The lack of such a service definition and well-defined API leaves the protocol as the only candidate as the invariant of the architecture: Without the NDN protocols, there is no architecture. A dilemma has arisen: The key property that makes NDN different—its protocol—is what needs to change; but changing it leaves nothing, because there is nothing else but the protocol that defines the architecture. Hence, the protocol needs to stay, and complex patches such as SNAMP are introduced to fix the fundamental flaws of NDN.

6.3.4 The Fundamental Abstraction

NDN relies on being able to send and receive interest and data PDUs between NDN nodes and between nodes and applications. In other words, NDN is dependent on the existence of an IPC mechanism to provide communication between the agents in the NDN architecture. In Jacobson et al. 2009, NDN builds on Ethernet, however, in concrete implementations such as CCNx and NDN-CCL, UDP is used. So, despite both Jacobson et al. 2009 and Zhang, Afanasyev, et al. 2014 claiming that content dissemination should be the fundamental abstraction of the future internet, they both require IPC in order to work.

6.4 Summary

Table 6.1 lists the taxonomy scores we have assigned NDN.

NDN is in many ways anti-extensional: The architecture is designed around the NDN protocol and its PDU format, with its focus on forwarding and naming. These low-level choices permeate the architecture and is exposed to applications via its low-level API that requires applications to construct, parse and validate individual PDUs. We therefore assign the lowest score of -2 to extensionality.

The low-level APIs do not encapsulate much: Protocols, PDUs formats, and name formats are all exposed. Transport mechanisms such as reliability and retransmissions are not encapsulated either, but is left up to the application. Even the connection concept does not exist, and if applications need persistent state spanning several PDU exchanges, they need to provide and maintain that themselves.

Without extensionality and encapsulation, there is nothing to abstract away and therefore no opportunity for generalisation—everything is exposed in its specialised, implementation-specific way. All this prompts us to assign the minimum score of -2 to generalisation and encapsulation.

NDN does not provide an indirection between the names used by and exchanged among applications and those used for routing and forwarding inside the network. Also, the same names are used to uniquely identify PDU destinations. There is no indirection in NDN, and we assign a score of -2 for this. Although SNAMP (Afanasyev, Yi, et al. 2015) introduces a layer of indirection, it is on a conceptual basis and has not found its way into any of the NDN implementations, and applications as well as network nodes need to agree on and be aware of it.

The choice to not apply any of the basic principles affects the properties of NDN.

Without extensionality, encapsulation, and generality, implementation choices and specialisations are exposed to the user applications of NDN. This leaves the architecture extremely rigid: Even small changes to data formats or protocol semantics requires changes in applications. We therefore assign a score of -2 to flexibility.

Without indirections, mobility is cumbersome because making mobility transparent is commonly done by using a level of indirection. By design, this is mitigated by mandating that mobility should be facilitated by dynamic routing tables. The interest mechanism provides such dynamic, transient, short-lived routing state for data PDUs which makes clients (or ‘consumers’) mobile, while the traditional long-lived routing state that routes

interest PDUs makes server (or ‘producer’) mobility difficult. We rate this asymmetric mobility property with a score of 0.

IA	Principles				Properties				
	Ext	Gen	Enc	Ind	Flx	Mob	Scl	Sec	Sim
NDN	-2	-2	-2	-2	-2	0	-2	1	0

Table 6.1: NDN Principles and Properties Scores.

Chapter 7

Discussion

There are no right or wrong ways to devise network architectures, but there are ways that allow us to reason about, understand and predict the behaviour of our networks; and there are ways that do not. This is the same in programming: Highly structured programming languages, such as functional languages, allow us to prove certain properties and exploit certain patterns precisely because they are so structured and—some would argue—restricted.

If we were to revert to assembly programming, we could do the same things quicker and dirtier; but we would have a hard time reasoning about, understanding, and predicting the behaviour of our programs. Most of us know this, and most of us happily use structured high-level languages because of this.

RINA is a network architecture that, if we follow its principles, allow us to say much more about our networks and their behaviours, and do much more with them, than any of the other architectures that we have analysed here. This includes TCP/IP which, despite its very fundamental and serious problems, is still taught as the correct way to do networking to students and programmers worldwide to an extent where if one tells them it is wrong, they might not believe it (Kurose and Ross 2013; Tanenbaum and Wetherall 2011).

The main role for any internet architecture is to support one or more communication services for applications. The amount of effort put into the definition of these services by the FIA literature is low; rather the effort is directed toward the implementation of the services (Day et al. 2008a,b; Goldstein et al. 2011).

7.1 APIs

We here discuss and compare the APIs of TCP/IP and RINA via a simple example covering the following scenario: A producer P makes available some data d which a consumer C

reads, changes to d' , and writes back to P .¹ This is shown in Figure 7.1. Where necessary, we assume that P and C understand two simple commands encoded in the SDUs of the PDUs: **GET**, which means “send me data”, and **PUT d** which means “save this data d ”. We show simplified commands and leave out some parameters to keep the example simple for both TCP and RINA. The RINA API used in these examples is specified by the prototype implementation of the RINA project IRATI (Vrijders et al. 2014).²

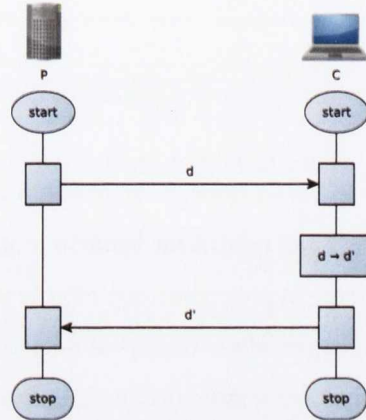


Figure 7.1: A consumer-producer API example.

We cover two scenarios:

Scenario 1: No Mobility P and C 's network addresses are static.

Scenario 2: Mobility P change network address after C have obtained d , but before it writes back d' to P .

For the TCP/IP case, we go through the example scenarios for TCP only. The UDP scenario would play out differently than TCP, because UDP sockets have different states and operational semantics. We do not consider it here. Also, we assume that the name N of P can be resolved to P 's address A using DNS.

7.1.1 Scenario 1: No Mobility

7.1.1.1 TCP/IP

Scenario 1 for TCP plays out as shown in Figure 7.2. P needs to register its name N with DNS out-of-band—there is no standard API for this, and it is in most cases a manual process. Afterwards, P creates a tcp socket, puts it in listen-mode to accept incoming connections, and spawns a new socket once C 's connect-command is received. C has beforehand looked

¹Technically, this is more than a simple producer/consumer scenario, but for simplicity we refer to it as such.

²<https://github.com/IRATI/stack/wiki/Application-API>

up N to find A and created a tcp socket, which it can now connect to P 's socket using P 's address A . C now sends a GET command to P , which interprets the command and returns d . C changes d to d' , and writes it to P with a PUT command. Afterwards, they both close their sockets.

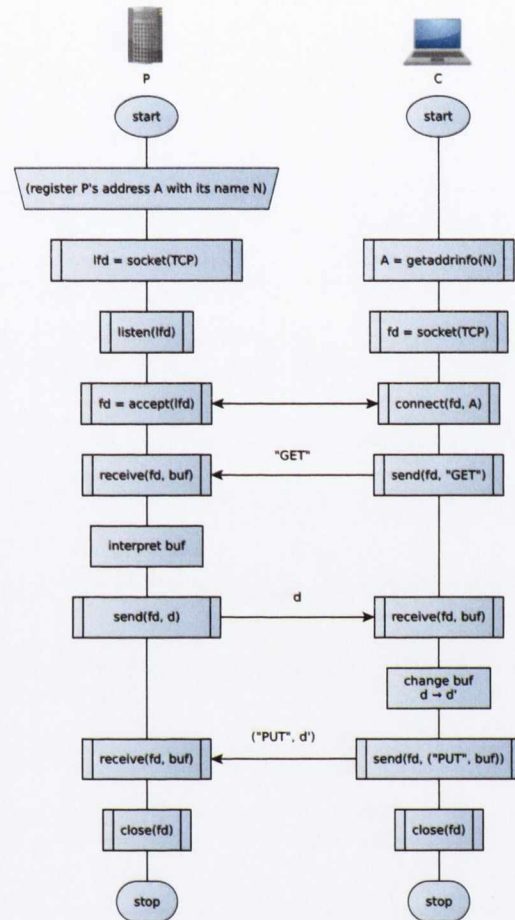


Figure 7.2: TCP producer-consumer example without mobility.

7.1.1.2 RINA

The scenario for RINA is shown in Figure 7.3. The RINA equivalent of a socket is a flow allocator instance (FAI), but its handle is a file descriptor, as with the socket API. P opens a listening FAI and registers its name N with the network³. Afterwards, it blocks while accepting incoming allocation attempts on the file descriptor. The consumer invokes an allocate command, specifying the producer's name N , to both obtain a file descriptor and synchronise with P . The rest of the scenario plays out similarly to the TCP case.

³Specifically, with a certain DIF. We assume that P and C share a DIF, and that this does not change for our example.

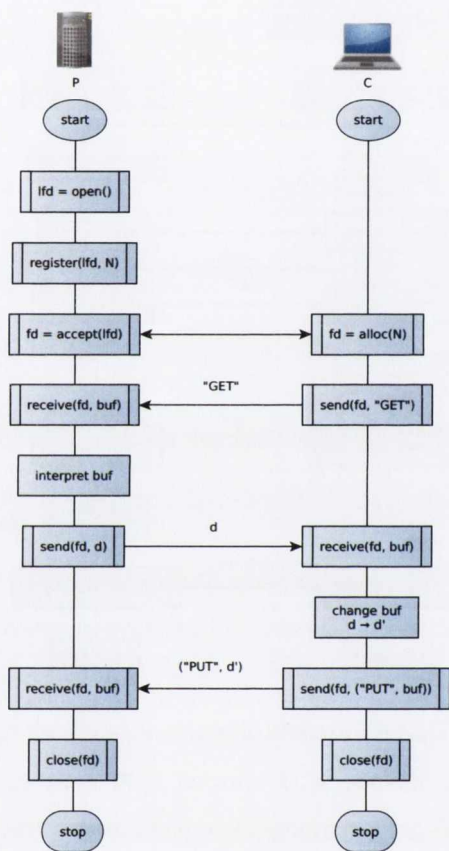


Figure 7.3: RINA producer-consumer example without mobility.

7.1.2 Scenario 2: Mobility

We now look at scenario 2, where the producer P changes its address after C has obtained the data d . Such an address change can be caused by P itself having moved in the network, or it can be caused by events out of the control of P , such as network reconfiguration or renumbering. Because an address change can be caused by outside events, P cannot always proactively mitigate it, but will in general need to be able to handle it at all times, lest its service will fail.

7.1.2.1 TCP

Scenario 2 for TCP is shown in Figure 7.4. The address change requires handling by both P and C . The scenario plays similarly to scenario 1 until the address change occurs. We assume that it can be detected by P when it happens. At that point, all P 's sockets become invalidated. P therefore needs to close both its listening socket identified with lfd , as well as its connection with C , identified with fd . It then needs to re-register with its new address A' . We assume that this can be done instantly and without noteworthy DNS propagation time, and that it can be automated.

P now needs to create a new socket, and put it in listening state. It can then accept incoming connection attempts from C .

C cannot immediately detect P 's address change—it will rely on communication attempts or socket connection state having to time out. Here, it attempts to write back d' to P , and when that attempt times out, it can guess that P has changed address. It closes the now useless connection, looks up P 's new address A' in DNS, and creates a new socket, which it connects to A' . On P the connection is accepted. The remaining data exchange can now happen as in scenario 1.

7.1.2.2 RINA

Scenario 2 for RINA is shown in Figure 7.5. Once P has registered its name N with the DIF used by P and C , the DIF will maintain the mapping, even if addresses change inside the DIF. The RINA API is designed to shield applications from network-specific information, so no operations require addresses as inputs—all operations work in terms of names. The flow between C and P created and controlled with the API operations, abstracts away the underlying network connections which are all managed by RINA. This is different from TCP, where the connections are exposed directly via the socket API.

All this means that the address change does not require involvement from neither P nor

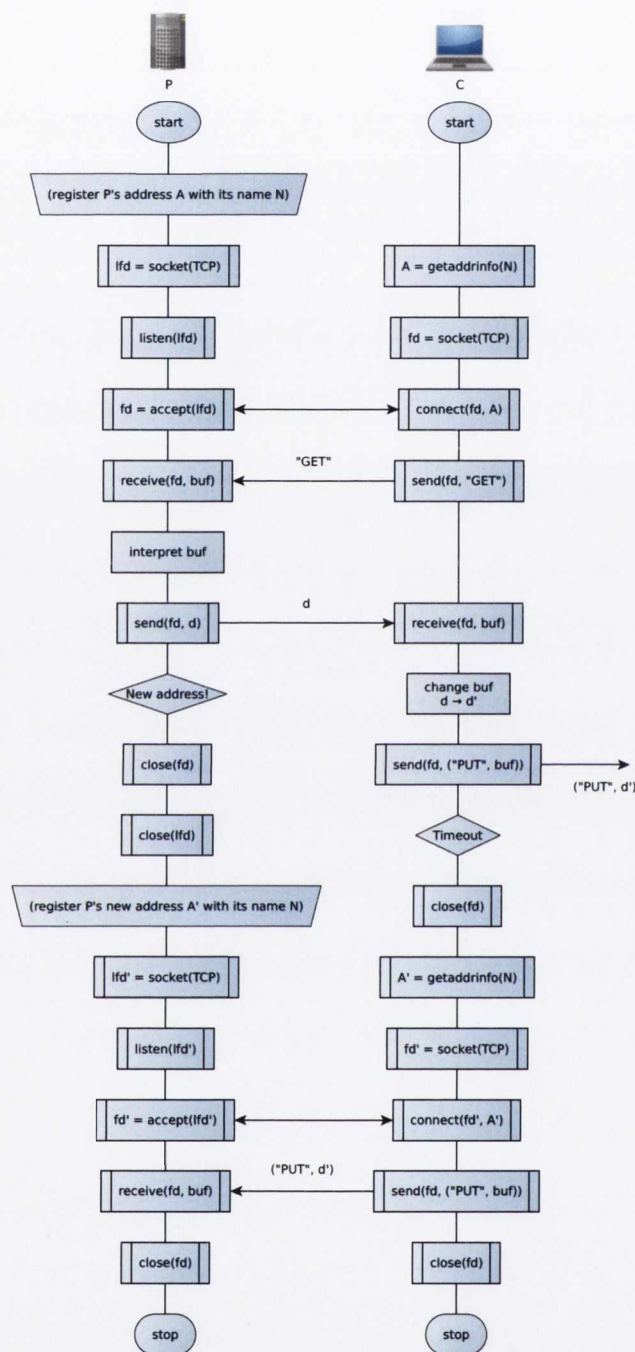


Figure 7.4: TCP producer-consumer example where *P* changes address after *C* has obtained its data *d*. *P* needs to re-register its name with a new address *A'*, and *C* needs to re-establish lost connections using the new address before *d'* can be written back.

C—it is handled entirely by the RINA framework.

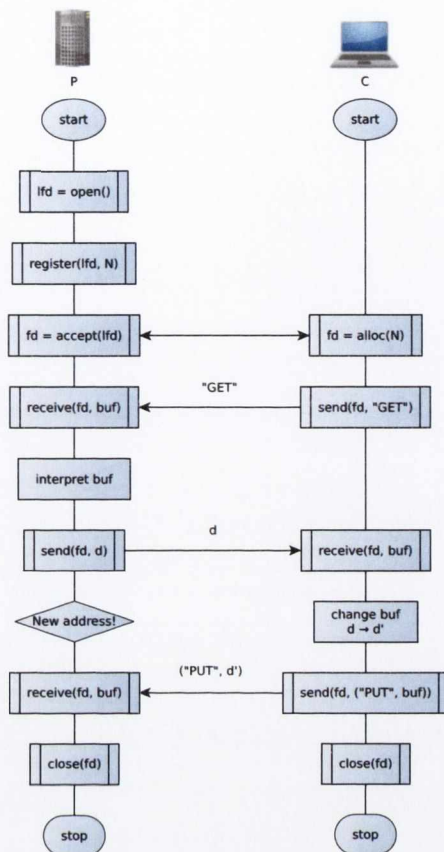


Figure 7.5: RINA producer-consumer example where P changes address after C has obtained its data d . In RINA, the address change is invisible to applications and is handled entirely by RINA. Therefore, the scenario is similar to Figure 7.3 (except for the address-change incident itself).

7.1.3 Comparison

Let us in Section 7.1.3 note some of the differences between TCP and RINA based on this example. In particular, we notice the following:

- RINA requires overall fewer operations than TCP in both cases.
- TCP depend on a name registration process that is not supported by the socket API. RINA has this built in.
- TCP provides a name lookup operation. In RINA such an operation is not needed, because no operations require addresses as inputs.
- The complexity the application increase with TCP when mobility needs to be considered, whereas there is no difference for RINA. This is because in TCP, connection and name

	TCP		RINA	
	Scenario 1	Scenario 2	Scenario 1	Scenario 2
No. of ops. (C)	7	12	5	5
No. of ops. (P)	7	12	7	7
Name reg. ops. in API	No		Yes	
Name lookup ops. in API	Yes		Not necessary	
Connection mgt.	Manual		Automatic (transparent)	
Name mapping mgt.	Manual		Automatic (transparent)	

Table 7.1: Differences between TCP and RINA for the producer-consumer example for both scenarios.

mapping management need to be handled by the application, whereas in RINA, it is handled by the framework.

This example illustrates that RINA supports address changes with no added complexity for the application, whereas an application running TCP needs extra complexity to handle it.

7.2 Architectural Trace

We will here trace the setup of state and transfer of data through the TCP/IP and RINA architectures for the examples from Section 7.1. We will use several definitions we have previously defined, for example protocol machine (PM), application process (AP), application entity (AE), endpoint identifier (EID), flow endpoint identifier (FEID), and connection endpoint identifier (CEID).

7.2.1 Scenario 1: No Mobility

7.2.1.1 TCP/IP

In TCP, the FEID—the identifier used in AEs for identifying other AEs, used to create a flow, i.e., a line of communication between AEs—and the CEID—the identifier used by the PMs to identify another PM, used to create a connection, i.e., a line of communication between PMs—are the same: it is a triple, consisting of an IP address, a protocol number for TCP, and a port number. This is because in TCP, there is no distinguishing between flow and connection. We will here use EID to refer to either. The EID in TCP/IP contains enough information so that it can be used for routing in the network.

First, we will go through the setup and binding of PMs in the network stacks on each side. This is depicted in Figure 7.6, where we will use the legend defined in Table 7.2. We will use the following labels for the components:

- *P* or *C*: The producer or consumer, respectively.

Component	Meaning
	Process—either the AP, or the OS process.
	A PM in the stack or an AE in the AP.
	Two PMs/AEs, locally bound to each other.
	Two PMs/AEs, remotely bound to each other.
	A layer delimitation in the stack.
	A relay for forwarding messages.

Table 7.2: The meaning of the components making up Figures 7.6 and 7.7.

- p_x : The AP of x (with x being either P or C , depending on if the process belongs to the producer or consumer, respectively).
- o_x : The OS process of x .
- a_x : An AE belonging to x .
- a_{xl} : An AE belonging to x for listening.
- s_x : A socket, i.e., a TCP PM belonging to x .
- s_{xl} : A listening socket belonging to x .
- i_x : An IP PM belonging to x .
- e_x : An Ethernet PM belonging to x .
- l_{xn} : Layer n on x .
- r_{xn} : A layer n relay belonging to x .

Initially, the situation is as shown in Figure 7.6a. Here, no sockets have been created, and the IP PMs (i_P and i_C) are connected to the same network and reachable from each other. The Ethernet PMs (e_P and e_C) are not necessarily on the same subnet.

In Figure 7.6b both P and C have each with a **socket** call created a socket (s_{Pl} and s_C)— P 's will become a listening socket—and corresponding AEs (a_{Pl} and a_C) are now present in the APs (p_P and p_C). The sockets are each bound to the corresponding AE, but at this point they are not bound to any local IP PM on either side and so they are not able to communicate.

With P 's call to **listen** in Figure 7.6c, P 's listening socket s_{Pl} is assigned a local EID thus bound to P 's IP PM i_P . It can now handle incoming connection attempts.

C needs to know the EID at which P 's listening socket is bound. This is usually found out via DNS in combination with conventions for well-known port numbers for specific services. In Figure 7.6d, P as called **accept** on the listening socket, and C has called **connect**. The two have been matched, because C used the EID with which s_{Pl} was locally bound. On P , a new AE (a_P) and socket (s_P) is spawned, which is immediately bound locally to the IP

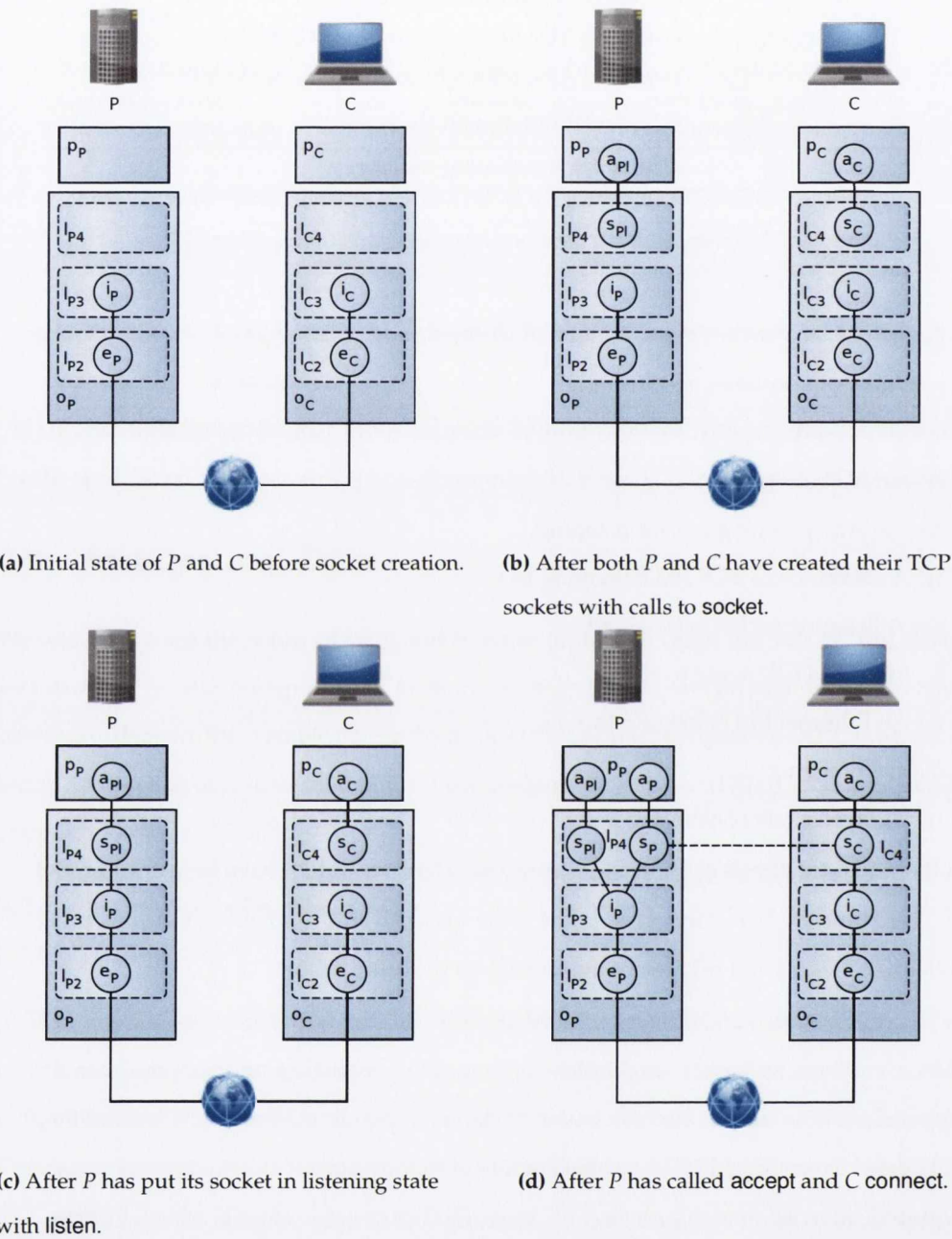


Figure 7.6: Creation of stack state in TCP/IP for producer P and consumer C of the example in Figure 7.2.

PM on a random port. On C , an AE (a_C) and socket (s_C) are created as well. The two sockets are furthermore bound remotely to each other. Once bound, either locally or remotely, a TCP socket cannot later be bound to something else (locally or remotely).

P and C can now exchange data. We have not detailed what the network looks like between C and P , so for this example we will assume that there is a router R capable of forwarding IP messages, and a switch S capable of forwarding Ethernet messages. This is depicted in Figure 7.7. Let us trace the transfer of the GET SDU from C to P via the 12 steps

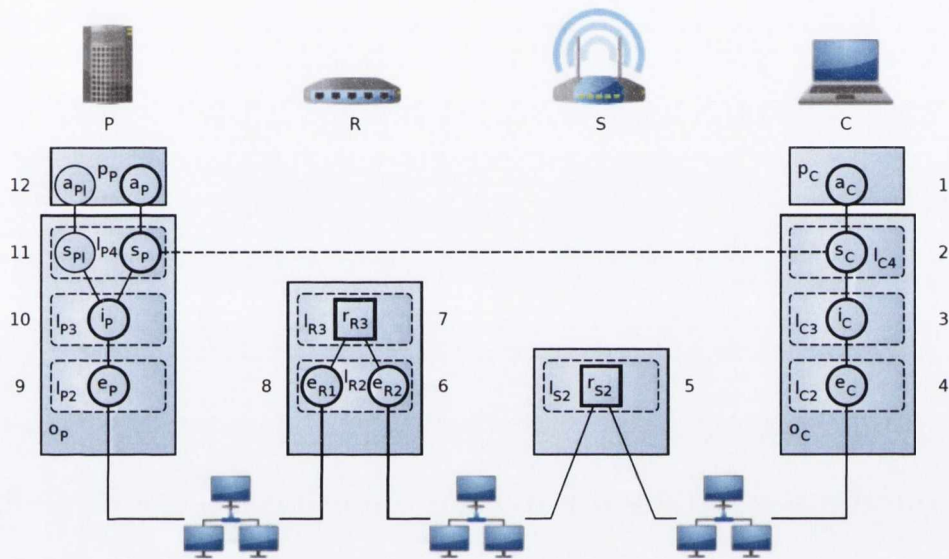


Figure 7.7: Transfer of data from C to P in TCP/IP. The PDU follows the path of the highlighted components.

in Figure 7.7, while we at each step show the current state of the PDU. Fields added in a step are highlighted with a thicker border, while fields removed in a step have a dashed border. The trace looks as follows:

1. `["GET"]`
The command "GET" is put in a buffer in C 's AE a_C . The buffer is transferred to the socket s_C via the `send` socket operation.
2. `[IP-i_c | IP-i_p | PN | port-s_c | port-s_p | "GET"]`
The socket s_C knows the source and destination EIDs, because these were defined when the socket was locally and remotely bound in the `connect` call. Each EID consists of an IP address, a protocol number (PN), and a port number. The source and destination ports are those of s_C and s_P , respectively, and the source and destination IP addresses

are those of i_C and i_P , respectively. These are prefixed to the SDU along with the protocol number for TCP.

- 3.

The IP PM i_C adds various IP control-fields to the PDU. These are not shown in this example, because they do not add any understanding at the level at which we are tracing the communication here.

- 4.

The Ethernet PM e_C adds an Ethernet address source and destination, as well as an ethertype (ET) field. The source and destination Ethernet addresses are looked up via the IP addresses with a combination of routing information and ARP, and are found out to be the addresses of e_C and e_{R2} , respectively. The ET field is known to be the ethertype for IP, because the PDU was received from the IP PM. The PDU is sent on the Ethernet link to S .

- 5.

In r_{S2} on S , the unmodified PDU is forwarded towards R based on its Ethernet PCI.

- 6.

Reaching e_{R2} on R , the PDU has reached its destination as defined by its Ethernet PCI. It is stripped of its Ethernet PCI, and passed to the IP relay r_{R3} .

- 7.

In r_{R3} , the PDU is relayed based on routing information in the relay and the IP addresses in the PCI. The result is that the PDU is passed down a layer to e_{R1} .

- 8.

In e_{R1} , an Ethernet PCI is prefixed to the PDU to allow its transfer to P . This is similar to the events in step 4, except that the Ethernet addresses in this case are those of e_{R1} and e_P . The PDU is sent on the wire towards P .

- 9.

At P , the Ethernet PCI is stripped at e_P , before the PDU is passed to i_P .

- 10.

At i_P , nothing of significance is stripped. The PDU is passed on to the socket s_P based on the destination port number in the PCI.

11. 

At the socket s_p , IP addresses, port numbers, and the PN is stripped, leaving only the SDU itself.

12. 

The SDU, containing "GET", is passed to P 's AE a_p in a buffer provided by P via its receive call.

The end result is, of course, that the SDU containing "GET" is transferred from p_c to p_p . On the way, various PCIs have been pushed and popped on the SDU: In s_c , IP and TCP PCIs are pushed—the IP PCI allows its transfer to i_p , and the TCP PCI allows it to reach s_p specifically (and not some other socket attached to i_p). In e_c , an Ethernet PCI is pushed, allowing the PDU's transfer to e_{R2} , where the PCI is popped, the PDU is relayed, before another Ethernet PCI is pushed, allowing the transfer to e_p .

We will not conclude further at this point. Let us examine how RINA carries out a data transfer before we compare the two approaches.

7.2.1.2 RINA

Let us now go through the scenario with RINA. RINA distinguishes between flows and connections, and by extension, between FEIDs and CEIDs. An application communicates via a FAI, which is equivalent to a socket in TCP in that it is the entity in the operating system associated with a flow, but importantly, it is *not* a PM itself—instead it redirects to one or more EFCP PMs which drive the connections of the flow. This means that a single flow in RINA is facilitated by possibly multiple connections. The FEID in RINA is based on the application name (AN) and identifies an FAI via the name of the associated AE; the CEID identifies an EFCP PM managed by the FAI. The CEID is never exposed to the applications—only the FEID is.

Let us trace the allocation of RINA entities in each stack. This is shown in Figure 7.8, where we use the legend in Table 7.3. We will use the following labels for the components:

- P or C : The producer or consumer, respectively.
- p_x : The AP of x (with x being either P or C , depending on if the process belongs to the producer or consumer, respectively).
- o_x : The OS process of x .
- a_x : An AE of x .
- f_{xn} : A FAI of x in layer n .

- e_{xn} : A EFCP PM of x in layer n .
- r_{xn} : An RMT of x in layer n .
- a_{xc} : An AE belonging to x for RINA control.
- c_{xn} : A RINA control entity of x in layer n .
- e_{xnc} : The EFCP PM belonging to a RINA control entity on x in layer n .

Initially, the situation is as shown in Figure 7.8a: the level 2 DIF consisting of d_{p2} and d_{c2} is established, facilitated by one or more level 1 DIFs. No connections between d_{p2} and d_{c2} have been created yet.

Figure 7.8b shows the situation after P has opened a control entity c_{p2} with the **open** call and registered the name of p_p with the level 2 DIF with the **register** call. Incoming allocation attempts for that name can now be handled.

When P calls **accept** and C **allocate**, FAIs in both ends, f_{p2} and f_{c2} , are allocated, associated with the respective AEs, a_p and a_c . EFCP PM may be allocated at this point—they will be bound locally to the FAI and the RMT, and remotely to each other (e_{p2} and e_{c2}). Figure 7.8c shows this situation.

C and P may now transfer data with calls to **send** and **receive**. Let us, similar to what we did in Section 7.2.1.1, trace the transfer of the “GET” message from C to P of Figure 7.3. This is shown in Figure 7.9, where we similar to the TCP trace introduce two relays R , for relaying layer 2 traffic, and S for relaying layer 1 traffic. The PDU transforms along the path from C to P as follows (the numbers in the list matching the numbers in Figure 7.9):

1. “GET”

In the AE, a_c , the SDU “GET” is put in a buffer and transferred to the RINA framework via a call to **send**.

2. “GET”

The SDU is received in the FAI, f_{c2} , bound to a_c . f_{c2} contains the FEIDs of the flow, which would enable it to trigger the creation of new EFCP PMs for new connections, if needed. In our example, a connection is already set up. The SDU is therefore forwarded to the one and only connection endpoint available, namely the EFCP PM e_{c2} .

3. CIED- e_{c2} / CIED- e_{p2} / “GET”

In e_{c2} , the SDU is prefixed with a PCI containing the CEIDs—in this example, it is the CEIDs of e_{c2} and e_{p2} , which are the source and destination, respectively, of the connection. The SDU is forwarded to the RMT r_{c2} .

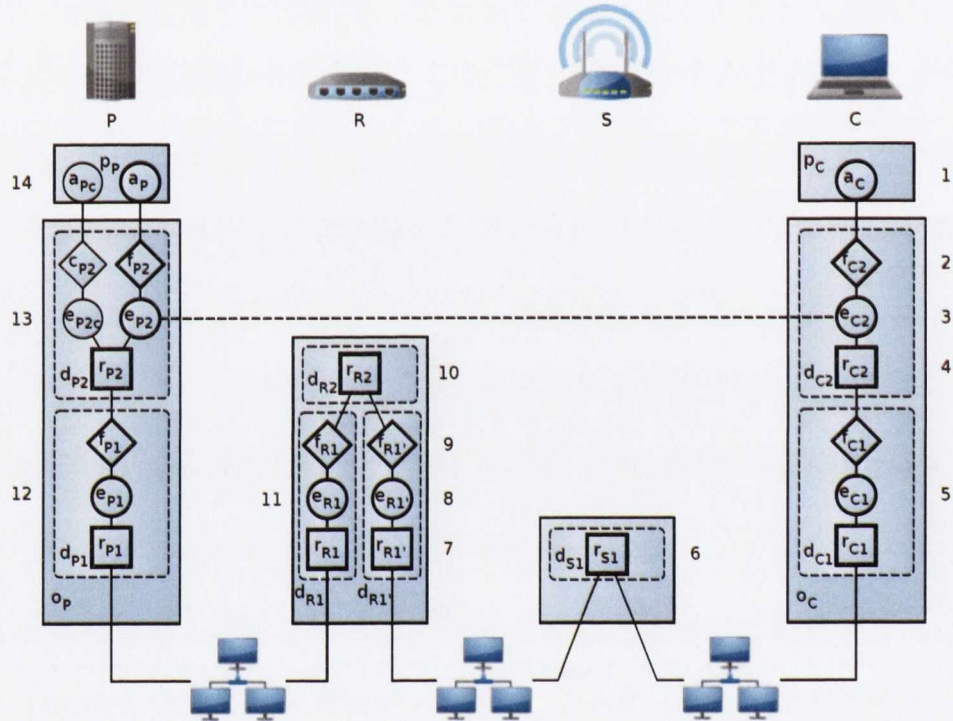


Figure 7.9: Transfer of data from C to P in RINA. The PDU follows the path of the highlighted components.

4. $\boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

When the PDU transfers from e_{C2} to r_{C2} , addresses of the DIPs in which the endpoints are located are added to the PCI. This allows intermediary DIPs to route the PDU. In this case, the addresses for d_{C2} and d_{P2} are inserted as source and destination addresses, respectively. The destination address is found by maintaining a mapping between CEIDs and addresses in the RIB of the DIP, along with mappings from destination AE names to DIP names, and from DIP names to addresses. That way an address can always be found from the destination AE name.

5. $\boxed{A-d_{C1}} / \boxed{A-d_{R1'}} / \boxed{CIED-e_{C1}} / \boxed{CIED-e_{R1'}} / \boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

In order to transfer the PDU from C to R via S in the DIF defined by $d_{R1'}$, d_{S1} , and d_{C1} , the PDU needs a PCI that makes sense to this DIF. This is prefixed in d_{C1} —specifically, CEIDs (for e_{C1} and $e_{R1'}$) and addresses (for d_{C1} and $d_{R1'}$) are added to the PDU, similar to what happened in d_{C2} in steps 2–4, but here for the lower layer. Finally, the PDU is sent on the wire towards S .

6. $\boxed{A-d_{C1}} / \boxed{A-d_{R1'}} / \boxed{CIED-e_{C1}} / \boxed{CIED-e_{R1'}} / \boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

In S , the PDU reaches r_{S1} , which looks at the addresses of the outermost PCI (to it, the inner PCI and the original SDU—the “GET”—are all part of an opaque SDU). The destination address of $d_{R1'}$ allows the PDU to be forwarded towards R .

7. $\boxed{A-d_{C1}} / \boxed{A-d_{R1'}} / \boxed{CIED-e_{C1}} / \boxed{CIED-e_{R1'}} / \boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

Arriving on R , the PDU enters $r_{R1'}$ in $d_{R1'}$. The destination address is found to be the current DIP, so the addresses are stripped from the PCI, and next, the destination CEID is interpreted. It matches $e_{R1'}$, so the PDU is forwarded there.

8. $\boxed{CIED-e_{C1}} / \boxed{CIED-e_{R1'}} / \boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

In $e_{R1'}$, the CEIDs are stripped, and the PDU is transferred up the stack to the FAI, $f_{R1'}$, which is unambiguous, because there can be only one FAI per EFCP PM (although, there can be multiple EFCP PMs per FAI).

9. $\boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

$f_{R1'}$ simply forwards the PDU to the AE with which it is associated, which in this case is the RMT, r_{R2} , of the DIP d_{R2} .

10. $\boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-e_{C2}} / \boxed{CIED-e_{P2}} / \text{"GET"}$

r_{R2} looks at the next level PCI, and relays according to the destination address of d_{P2} , which means the PDU is forwarded to f_{R1} of d_{R1} .

11. $\boxed{A-d_{R1}} / \boxed{A-d_{P1}} / \boxed{CIED-\Phi_{R1}} / \boxed{CIED-\Phi_{P1}} / \boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-\Phi_{C2}} / \boxed{CIED-\Phi_{P2}} / \boxed{"GET"}$

In d_{R1} , a lower-level PCI is prefixed to allow the PDU to travel to P , the wire to which it is forwarded on.

12. $\boxed{A-d_{R1}} / \boxed{A-d_{P1}} / \boxed{CIED-\Phi_{R1}} / \boxed{CIED-\Phi_{P1}} / \boxed{A-d_{C2}} / \boxed{A-d_{P2}} / \boxed{CIED-\Phi_{C2}} / \boxed{CIED-\Phi_{P2}} / \boxed{"GET"}$

Reaching P , the PDU passes through d_{P1} , where the lower-level PCI is stripped on the way, before being handed up to d_{P2} . This is similar to steps 7–9.

13. $\boxed{A-d_{R1}} / \boxed{A-d_{P1}} / \boxed{CIED-\Phi_{R1}} / \boxed{CIED-\Phi_{P1}} / \boxed{"GET"}$

Again, the next-level PCI is stripped similar to in step 12, and now only the original SDU containing "GET" is left. It is passed up the stack to a_P via a buffer.

14. $\boxed{"GET"}$

In a_P , the message is received via a call to `receive`. The transfer is complete.

Similarly to what happened in Section 7.2.1.1, the result is that the SDU "GET" is transferred from p_C to p_P , and various PCIs have been pushed and popped along the way in a similar fashion: The first PCI pushed on C in step 3 and 4 consists of a set of addresses and ports and allows the forwarding in step 10 on R towards P (based on the addresses) and the forwarding in in step 13 from r_{P2} to e_{P2} (based on the CEIDs). The second PCI pushed in step 5 allows the forwarding in step 6 on S towards R , and the forwarding in step 7 from $r_{R1'}$ to $e_{R1'}$.

7.2.2 Scenario 2: Mobility

We will now examine what happens when P changes address. For TCP/IP, the IP address change, while in RINA, the level 2 address (i.e. the address of d_{P2}) change.

7.2.2.1 TCP/IP

In TCP/IP, an address is the primary identifier for an IP PM. The scope of an IP address is all of L3 (except if NAT is in use, which we will ignore here). It is also exposed to other layers and applications. The address is part of the state of the IP PM and of L4 PMs. It is also part of the application state. If the address change, all these components will need to handle it.

A change in P 's IP address is equivalent to the existing IP PM being terminated, and a new IP PM being created with a new address. This affects other PMs locally and remotely and require recovery on both P and C . The process is shown in Figure 7.10, where the following takes place:

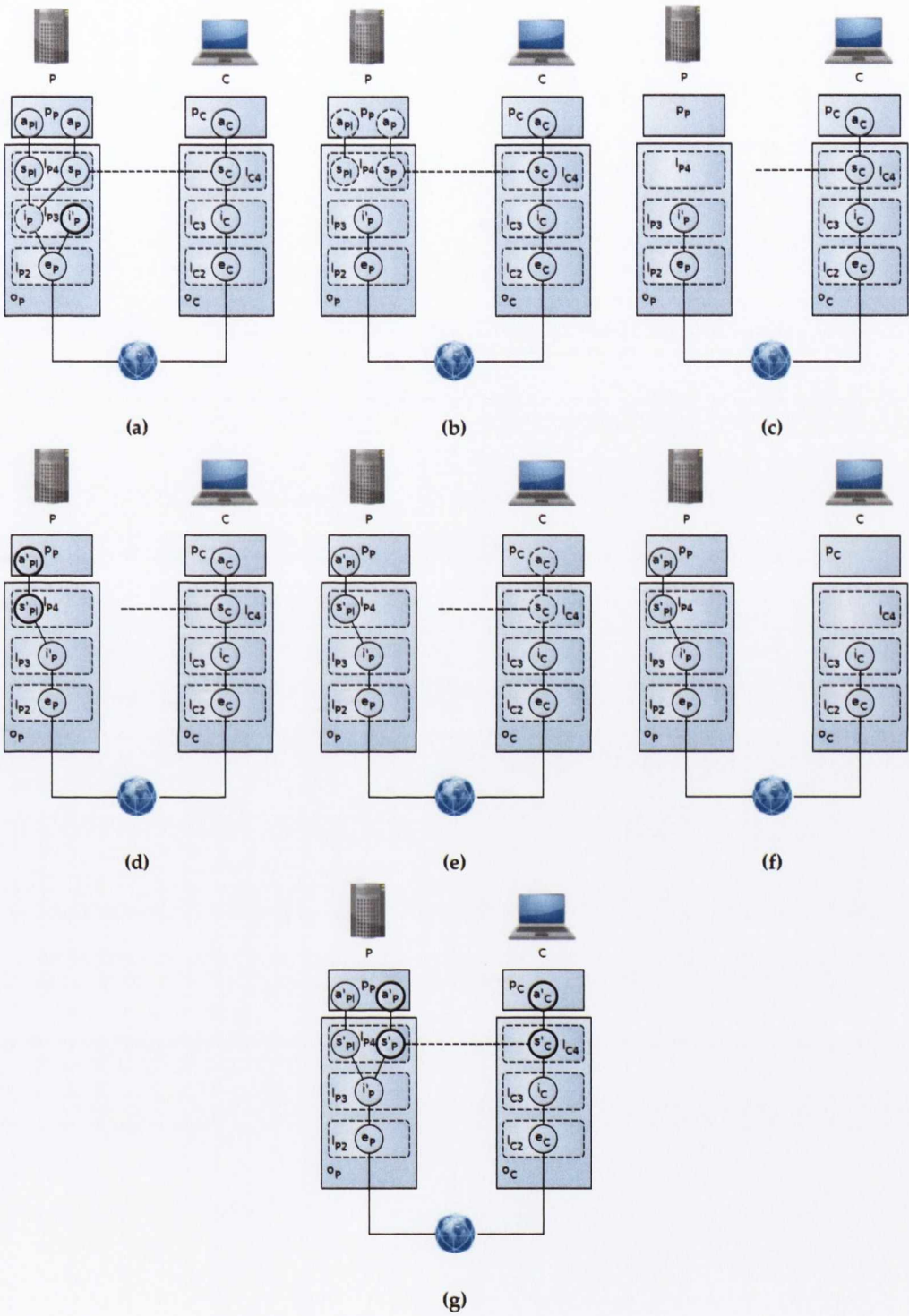


Figure 7.10: Effects of IP address change in TCP/IP including recovery. Components with dashed borders are being torn down, while components with highlighted borders are being created. The process is explained in details in the text and omitted here.

Figure 7.10a The IP address change is equivalent to a tear-down of the existing IP PM, i_p , and the creation of a new one, i'_p .

Figure 7.10b The already bound TCP PMs cannot be rebound to the new i'_p without resetting their connection state, because the state of the connections depend on the EIDs, of which the IP addresses are a part. For this reason, the socket API does not allow rebinding TCP sockets. Instead, they become invalidated. The application at P can detect it via error codes from the socket API, and the sockets, s_{pl} and s_p , and associated AEs, a_{pl} and a_p , are torn down.

Figure 7.10c All state needed to support data transfer has now disappeared from P . C 's state still exists— s_C still believes it is remotely bound to the now non-existent s_p .

Figure 7.10d On P , state now needs to be rebuilt. p_p creates a new listening socket s'_{pl} and its associated AE, a'_{pl} . The socket is bound locally to i'_p with the new IP address. P now accepts incoming connection attempts on the new IP address.

Figure 7.10e C discovers, possibly by means of a timeout, that the peer to s_C is gone, and initiates a tear down of the socket, s_C , and its associated AE, a_C , because the socket cannot be rebound and a new one has to be created instead.

Figure 7.10f C now has no state left to support data transfer.

Figure 7.10g C recreates state by creating a new socket, s'_C , and an associated AE, a'_C . s'_C connects to s'_p on P using the new IP address. Here we assume that C can obtain the new IP address. The state has now been rebuilt, and data transfer may be resumed on the new connection.

We notice how a change of IP address on P leads to a reconstruction of state entities at both P and C in L4 and layer 5 (L5), and in L3 on P as well.

Let us see how RINA handles a change of address.

7.2.2.2 RINA

In RINA, an address is a secondary identifier of a DIP. The primary identifier is the name of the DIP—the address is merely a shorthand synonym for the name, used locally within the DIF, which allows it to be tailored to the DIF, for example with regards to format, namespace and the kind of graph or routing information it contains. The scope is the local DIF; it is not visible outside of the DIF. Locally, the address can be found by the RMT in the RIB of the DIP, but it is not part of the state of any of the EFCP PMs—their state relies on CEIDs only.

It is not part of the state of the FAI nor of the AE, either. If the address change, only the RIB and by extension the RMT need to handle it locally.

To compare with TCP/IP, we let the address of d_{p2} of Figure 7.9 change. The address change is equivalent to adding an additional address to d_{p2} —so it has two—followed immediately by the deletion of the old address—so it once again has only one address.

The addition of the extra address will trigger an update of the RIBs of the layer which d_{p2} belongs to, specifically the RIBs in d_{p2} , d_{R2} , and d_{C2} , so that the RMTs of these DIPs may forward PDUs with the new address.

The deletion of the old address will trigger a similar update, removing the old routing information from the DIF. Importantly, the address change does not require tearing down the DIP—it lives on, with its new address. The only change in the 14 step process starting on page 131 associated with Figure 7.9 is in step 4, where the PDU needs to be prefixed with the new address. The mapping in the RIB, which maps the destination CEID to the destination address is now outdated, so we need to first look up the name of the destination AE to find the name of the destination DIP, and then use that to find its DIF-internal address.

The internal connection state of the EFCP PMs e_C and e_P do not depend on the addresses, only on the CEIDs—but these are unchanged.

In summary, the address change in RINA does not require tearing down PMs and the subsequent recovery as it did in TCP/IP. This, in turn, shields the applications from having to rebuild their state in response, as it happened with TCP/IP. In fact, address changes in RINA are invisible to the applications.

7.2.3 Summary

While both TCP/IP and RINA share a layered structure where PCIs are pushed and popped systematically as we saw in Sections 7.2.1.1 and 7.2.1.2, there are notable differences:

1. The PCIs in TCP/IP differ from layer to layer: L4 PCI contains ports, L3 PCI contains addresses and protocol numbers, L2 PCI contains Ethernet addresses and ethertype identifiers. In RINA, the PCI content is similar in all layers: All layers contain a set of addresses and CEIDs.
2. TCP/IP uses special identifiers to specify the protocol in use in the layer above: In IP, the protocol number specifies the L4 protocol in use, and in Ethernet, the ethertype field specifies the L3 protocol in use. In RINA, no such identifiers are used.

Differences	
TCP/IP	RINA
Different PCIs	Similar PCIs
Protocol identifiers (PMs/ETs)	No protocol identifiers
One set of identifiers per layers	Two sets of identifiers per layer
Addresses denote L2/L3 PMs	Addresses denote DIPs within layer
Ports denote L4 PMs relative to an L3 PM	CEIDs denote EFCP PM relative to RMT
Change of address require PMs to be recreated	Change of address does not require PMs to be recreated
Change of address affects applications and require recovery	Change of address is invisible to applications
Similarities	
Layered structure	
Push/pop PCIs	

Table 7.4: Differences and similarities between TCP/IP and RINA evident in the traces of Section 7.2.1.

- 3. In TCP/IP, there is one set of identifiers (addresses or ports) per layer, while in RINA there are two sets of identifiers per layer (addresses and CEIDs).
- 4. In TCP/IP, an address denotes a single PM in a layer. In RINA, an address denotes a DIP in a layer, which may contain several PMs.
- 5. In TCP/IP, a port denotes a PM in L4 relative to the PM in L3 it is bound to. In RINA, a CEID denotes a EFCP PM relative to the RMT of the same DIF.
- 6. In TCP/IP, a change of address affects both the state of the stacks, requiring PMs and sockets to be recreated and bound anew to each other and to the application, as well as the state of the applications, which need to recreate and reinitialise its socket handles. In RINA address changes affects the stack, but only minimally in that existing stack entities may adapt to the change in place, while applications are unaffected.

We have summarize our findings of our architectural traces in Table 7.4.

7.3 Evaluation of TCP/IP and RINA

We have now seen both the RINA and the TCP/IP architecture. Below, we summarise how they are different.

7.3.1 Connection Management

In TCP/IP, APs manage each connection via the socket PM abstraction and its socket API. In RINA, PMs are managed by the DIF. This has the following consequences:

- The API in TCP/IP differs depending on the protocol of the PM. For example, the API controlling a TCP socket differs from that controlling an UDP socket. The AP explicitly chooses the protocol and by that, the API. If protocols change in a way that affects the PM API, or if different protocols are desired, the AP must be updated.

The API in RINA does not differ depending on protocol. The AP does not choose a protocol, but instead choose what QoS is required; the DIF then makes adequate choices to meet these requirements, depending on the service provided by lower layers. Changes in protocols, whether or not an update to existing protocols or introduction of new protocols⁴, affect the DIF, which might need updating, but no application will need to change.

- In TCP/IP, there are eight operations: open, send, receive, close, bind, listen, accept, and connect. The latter four are necessary in TCP/IP to be able to control the connection or spawn new connections (by explicitly spawning a new socket). In RINA, they are superfluous because connections are managed by the DIF. Therefore, the RINA API is simpler than TCP/IP's socket API, with only four operations: allocate, deallocate, send, receive.
- In TCP/IP, a socket is statically bound one-to-one with the handle in the AP. In other words, once a handle is created, it is fixed to that particular socket for the remainder of its life time. If the socket becomes invalidated for any reason, the AP will need to close it and open a new one. Multiple connections require multiple sockets and therefore multiple AP handles, so any multihoming has to be coordinated by the AP.

In RINA, multiple PMs can be dynamically bound to the AP handle. If a connection becomes invalid, the DIF can deallocate the invalid PM and create a new connection and associated PM. This is transparent to the AP, who does not need to manage connection failures (it does need to handle flow failures, i.e., if no connection is available at all). Multihoming is also transparently handled by the DIF.

7.3.2 Endpoint Identification and Binding

In TCP/IP, the connection endpoint and the flow endpoint are the same and manifested as a socket. The connection EID used in the TCP protocol is a tuple consisting of an IP to identify

⁴New protocols should not be necessary in RINA due to the flexibility of EFCP.

the network PoA, and a protocol and port number, to identify the specific socket. This forms a path-name specifying the path through the stack leading to the socket. This EID is also used to identify sockets by other APs outside of the protocol.

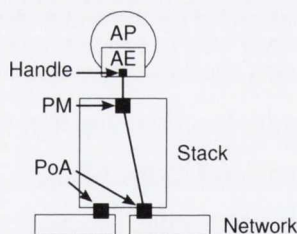
In RINA, the FAI is the flow endpoint. It is identified uniquely using the name of the AE it is bound to. Each connection's EIDs is a random identifier, unique within the DIP. The EID is bound to a specific PoA using the DIP's Forward Information Base (FIB).

These choices have the following consequences:

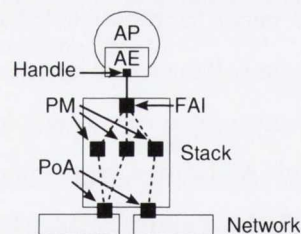
- Static versus dynamic per-connection PoA:

In TCP/IP, the socket is bound via its EID to a particular PoA. Therefore, a socket cannot change PoA (or port number). If the PoA changes, for example, if the PS or whole network is renumbered or change location, or if a new EID is needed because of sequence number roll-over, the socket becomes invalidated. As we saw in Section 7.3.1, connections are managed by the AP. Therefore, the invalidation has to be detected and handled by the AP.

In RINA, the FAI manages PMs, and each PM is bound to an EID, which is mapped dynamically to a PoA in the FIB. A connection may change PoA without the PM becoming invalidated as long as the FIB is updated. If the PM becomes invalidated for other reasons, such as sequence number roll-over, the FAI will detect and handle invalidations without it affecting the AP.



(a) A TCP/IP stack and application. Applications bind statically to the socket/PM, which binds statically to a PoA. If either the PM or PoA becomes invalidated, the application needs to recreate the bindings.



(b) A RINA stack and application. Applications bind statically to the FAI, which maps dynamically to PMs. The PMs bind dynamically to PoAs. The stack handles invalidations of both PoAs and PMs, and dynamically remaps bindings.

Figure 7.11: Connection management and endpoint bindings in TCP/IP and RINA

7.3.3 Application Names and Addresses

We here define human-friendly names as names that contain a description of the things they name.

In TCP/IP, there are no human-friendly names. Instead, DNS has been devised as a service that resolves human-friendly domain names to the IP-addresses needed to build the EID tuples used in TCP/IP. The applications perform this name resolution in a separate step before using the socket API to establish connections. The EIDs used as inputs to the socket API have a structure that reflects the structure of the network stack, and the IP addresses used in the EIDs have a structure that reflects how IP does routing.

In RINA, flows are created using human-friendly names, and a separate name resolution step is not needed. Mappings from the human-friendly application names to names and addresses that are more useful for a particular layer are encapsulated in each layer without involvement from anyone outside of the layer. No name or address that reflects anything about the inner workings of the network is exposed.

This has the following consequences:

- Unpluggable or pluggable NRS

In TCP/IP, name resolution is handled by the application. If the name resolution service changes, for example, if its input or output format changes, the applications will have to change too.

In RINA, name resolution is handled by the network stack. The name resolution service or its inputs and outputs can change without it affects applications, as long as applications' input format to the RINA API is unchanged.

- Exposed or encapsulated names, addresses and ports

In TCP/IP, the API exposes the EID with its inherent semantics, because it is used as socket identifier. If these semantics change, applications need to change too. Therefore, these cannot change unless there is global agreement to do so, which means that changing port numbers, protocols numbers, and IP formats and semantics also requires global consensus. The still ongoing change from IPv4 to IPv6 is one example of how this is problematic. The exposed semantics contained in the EID also increase possible attack vectors. For example, well-known ports risks exposing systems to attackers.

In RINA, the human-friendly names used to identify sockets contain no semantics originating from the network. Addresses and EID are encapsulated in each layer. Therefore, if the semantics of a layer's addresses or EIDs change, only the layer itself

need to consider the repercussions; applications or other layers need no updates. Attack vectors are reduced when the attacker have no knowledge of internal layer semantics at his disposal. In RINA, well-known ports do not exist, and the range of typical EID namespaces makes guessing or brute-forcing them ineffective.

7.3.4 Layers and Protocols

In TCP/IP, there is in principle a fixed number and ordering of layers. Layers are distinguished by their functions, and they are facilitated by different protocols. L2, L3, and L4 each expose one or (in the case of L4) several different socket API to applications, but with the kernel-space/user-space boundary in the POSIX API specification, they are not meant to be used as inter-layer APIs.

In RINA, the number of layers is variable. Layers are distinguished by their QoS and scope, but provide the same function, IPC, facilitated by the same protocol, EFCP. All layers expose the same API, used by both applications and other layers.

This has the following consequences:

- Allowed layer combinations: In TCP/IP, compositions of layers other than the standard five-layer composition goes against the grain of the model. That there is a need to compose layers in new ways is evident when we look at the typical layering in core networks where the number of layers is ten or more. Confusion arises when trying to talk about 'layer 3' in a scenario where there are multiple IP layers, or when a new protocol such as MPLS enters the field and is described as belonging to 'layer 2.5'. These are symptoms that the model diverges from real world needs. In RINA, layers are meant to be composed, and layers are numbered relative to each other.
- Practical layer combinations: In TCP/IP, due to the problems of inter-layer APIs mentioned in Section 4.4.2, layer composition requires customised glue-code at a low abstraction level, typically using PDU (or PDU pointer) queues. Because protocols generally differ from layer to layer, headers need to be prepared by the layer that enqueues the PDU to match those of the layer that dequeues them. The enqueueing layer thus need to know the type of the dequeuing layer before the PDU is enqueued. Any new combination of enqueueing and dequeueing layer require new glue code. The glue code requires translation of addresses from one layer to another. This requires a custom mapping service between those two specific layers, including mechanisms to keep such a mapping up to date. In RINA, the standardised APIs allow compositions without glue code: The user of a layer's API can be an application as well as some

higher layer, and each layer interacts with lower layers using the API of the lower layers. The API ensures that headers are created by the same layer that need them. The FIB of a layer in combination with the lower layer's RIB together provide the necessary address mappings. They are kept updated via mechanisms that are already present.

- **Protocols:** In TCP/IP, each protocol typically provides a fixed QoS. Ethernet, MPLS, IP, TCP and UDP all provide data transport, but differ in routing mechanisms, scope, degree of support in other layers and applications, and service guarantees. A change in QoS requirements can lead to a change in protocol, requiring comprehensive network and system upgrades. The one-QoS-one-protocol mindset means that the number of protocols in and surrounding TCP/IP is large.

In RINA, EFCP can handle a wide range of scopes and QoS. A change in QoS requirements require reconfiguration of QoS parameters for existing protocols, but does not require roll-out of new protocols. Customised routing or forwarding mechanisms, similar to, for example, that in MPLS, can be implemented in one layer without changing neither its protocol nor other layers.

7.3.5 Routing and Renumbering

In TCP/IP, routing is done on IP addresses, which are assigned to PoAs and exposed and used by upper layers, including the transport layer and applications.

In RINA, routing is done on addresses internal to the layer, which are assigned to the DIPs of the layer.

Routing table sizes are bounded by the number of addressable entities. For a network with N nodes and E edges, there are $P = 2E$ PoAs. In general, $\frac{N(N-1)}{2} \geq E \geq N - 1$, so $N(N - 1) \geq P \geq 2(N - 1)$.

Routing table entries can be aggregated, if the topology of the address-space correlates to that of the network graph, i.e., if entities that are near each other in the network graph have addresses that are near each other in the address space (for some definition of nearness).

This has the following consequences:

- **Routing table size upper bound**

In TCP/IP, routing table sizes are bounded by number of PoA. In RINA, routing table sizes are bounded by the number of DIPs, i.e., by the number of nodes in the network graph. At best, routing table sizes in TCP/IP are bounded to be double the size of that in RINA. At worst, the bound is the square of the bound in RINA.

- **Renumbering**

In TCP/IP, renumbering PoAs invalidates all connections using those PoAs. Applications will have to handle any connection failure and additionally detect—by a prolonged timeout—that another application’s address has changed before probing DNS for an updated address.

In RINA, renumbering PoAs does not invalidate existing connections, if the FIB is updated to reflect the change, which will happen, if incoming PDU of existing connections have new source addresses. For connections where the FIB entries are not updated, the FAI detects the connection failures, and create new ones after having looked up the new address in the RIB. Applications are therefore not affected by renumbering.

- Routing entry aggregation

In TCP/IP, keeping the addresses aligned with the network graph to allow for effective aggregation is difficult because renumbering disturbs users of the network and might lead to application failures. The address format cannot change without extensive changes to APIs and applications, and without requiring global coordination, so addresses in other formats cannot in practice be used, even if they are more appropriate for aggregation.

In RINA, addresses can be kept aligned with the network graph because renumbering can be done without disturbances to the users of the network. Because addresses are not exposed outside of each layer, address formats can be changed to allow for better alignment with the network’s topology resulting in more effective aggregation.

7.4 What to Address

We have seen that TCP/IP and RINA choose to assign addresses to two different entities: PoAs and processes, respectively. CCN originally addressed content chunks, but in its NDN incarnation allow addresses to identify various types of entities, for example, routers or networks (Zhang, Afanasyev, et al. 2014). The choice of what to address will define important properties of an architecture. We illustrate this with an abstract example using a graph consisting of edges and nodes. A PoA is where an edge and a node meet. We here walk through what it means to address PoAs (the IP approach) as opposed to nodes (the RINA approach).

7.4.1 Addressing PoAs

Consider a graph as depicted in Figure 7.12. It has four nodes, *A* through *D*, and is fully connected. Each PoA in the graph is assigned a number (1 through 10). At each of the nodes, a routing table contains information about where to forward PDUs to based on the destination address of that PDU. For example, the routing table for *A* reveals that if a PDU with a destination address with value 9 needs to be forwarded, it should be sent out on the PoA with address 1.

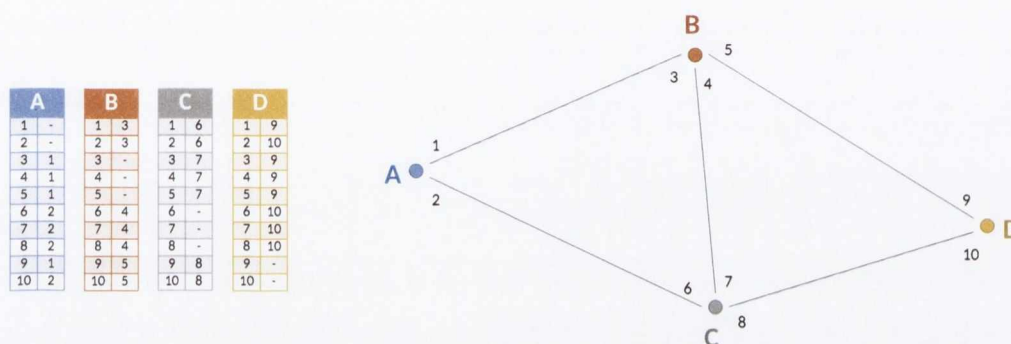


Figure 7.12: Example graph showing nodes (*A* through *D*), PoAs (1 through 10), and the nodes' routing tables.

Consider now what happens when we want to route a PDU from *A* to *D*. Because the routing table entries are defined in terms of addresses, the first step is to translate the destination *D* to an address. Let us say that this results in the destination address 9.⁵ Similarly, the source address is resolved to 1.

The routing now progresses as follows (shown in Figure 7.13):

1. On *A*, the destination address 9 is looked up in *A*'s routing table. This results in the PDU being forwarded on PoA 1, which leads to *B*.
2. Similarly on *B*, the forwarding PoA is found out to be 5, which leads to *D*.
3. Finally, on *D*, the value for address 9 gives a nil-value, indicating the PDU has reached its destination.

Let us now see what happens, if a connection goes down—illustrated in the following example by an edge disappearing and routing tables updating mid-transit. Concretely, let us illustrate what happens, if edge *BD* goes down while the PDU is on transit between *A* and *B* (shown in Figure 7.14):

⁵Another valid result would be 10.

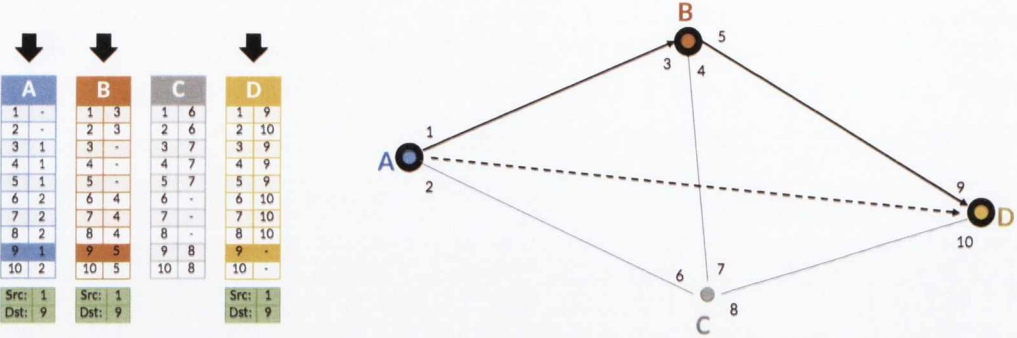


Figure 7.13: Routing example for a PDU travelling from A to D with destination address 9.

1. On A, the PDU is forwarded on PoA 1 as before, leading it towards B.
2. The connection between B and D breaks down. In our graph, this means the edge BD disappears. It also means that the routing tables gets updated. All entries containing the addresses of the PoAs belonging to the now non-existent edge are removed from the routing tables.⁶ Also, some entries not related to the broken connection are updated to route around the failure. Unfortunately, this means that all entries with the address 10 are removed, which happens to be the destination of our PDU in transit!
3. The PDU reaches B. B's routing table no longer contain an entry for destination address 9. B therefore has no choice but to drop the PDU, even when we know that D could be reached by routing the PDU to destination 10! The problem is that the network does not knows this—all it knows are the destination and source addresses of the PDU and the contained routing information of the nodes.

For this kind of failure, the PDU will be dropped. After a timeout period, the PDU towards D can be retransmitted, this time with an updated destination address of 10.

7.4.2 Addressing Nodes

Let us consider addressing the nodes instead. We still wish to transmit a PDU from A to D. Because we use node addresses, D already serves as the destination address, and can be used directly in the PDU. The routing tables will contain entries for node addresses instead of PoA addresses. This is shown in Figure 7.15, and the transmission will progress as follows:

⁶We show the routing tables with the state at the time of forwarding—therefore, in Figure 7.14, tables B, C, and D are updated, while A shows its state before connection breakdown.

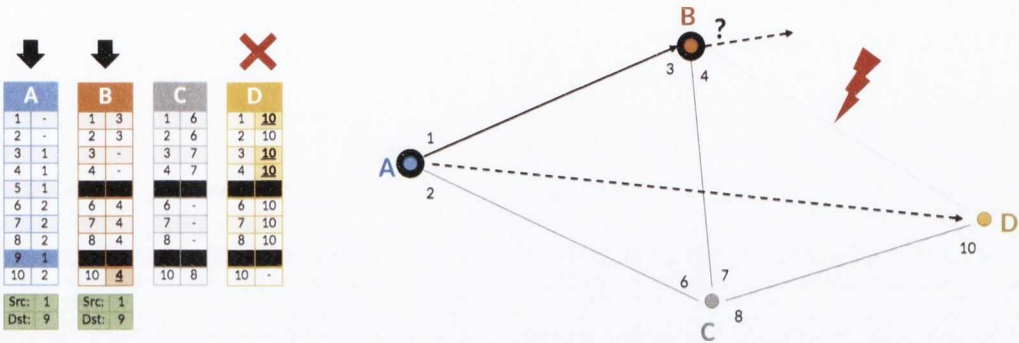


Figure 7.14: Routing example for a PDU travelling from A towards D with destination address 9 with a connection-failure mid-transit. Routing table A shows its state before breakdown, while B, C, and D shows their state after breakdown, so that they all show their state at the time of forwarding. Removed entries are shown with black, while updated entries are shown with underlined values. The PDU will be dropped at B because its destination address points to a PoA that has disappeared from the network and routing tables.

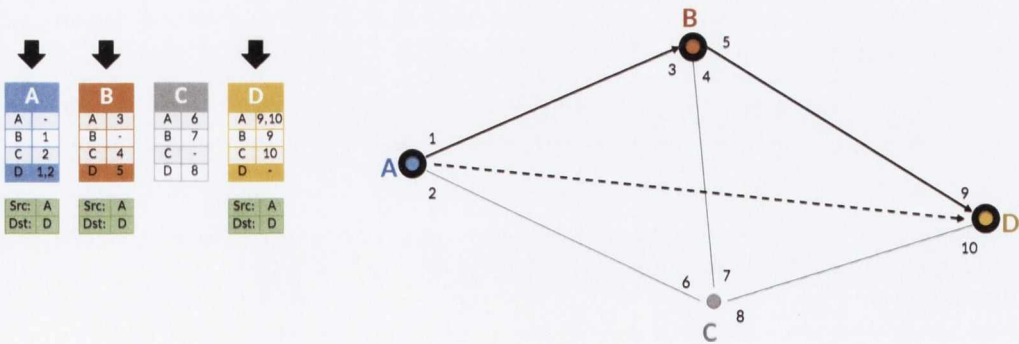


Figure 7.15: Routing on node addresses.

1. On *A*, the destination address *D* is looked up and the PDU is forwarded to PoA 1, leading to *B*.
2. On *B*, a similar lookup forwards the PDU on PoA 5, leading to *D*.
3. Arriving at *D* the destination has been reached.

There is no significant difference in the routing process compared with the routing-on-PoAs-example, apart from the fact that we are routing on node addresses.

Let us again examine the situation when the connection between *B* and *D* goes down while the PDU is in transit between *A* and *B*.

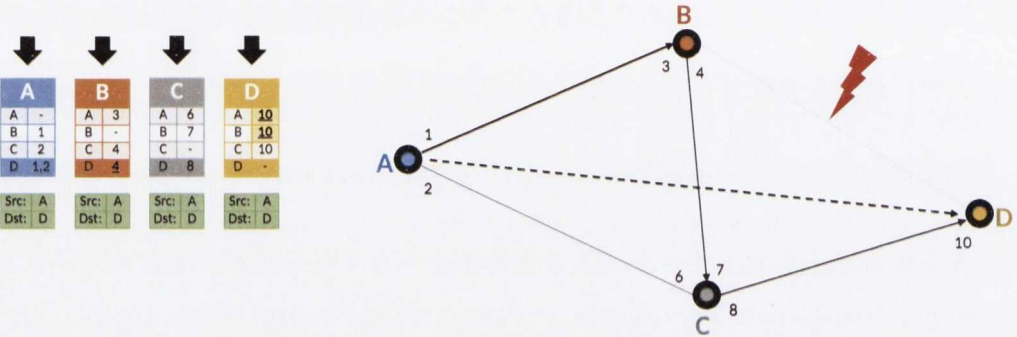


Figure 7.16: Routing on node addresses for PDU travelling from *A* to *D* with a connection-failure mid-transit. The PDU will be re-routed and will reach its destination.

1. At *A*, the PDU is forwarded to PoA 1 leading towards *B*.
2. Connection *BD* breaks down. Routing tables are updated. No entries are removed, because no node has been removed from the network—however, values are updated to reflect the new routes (shown with underlined in Figure 7.16).
3. At *B*, the updated routing table is used to determine the forwarding PoA, and the PDU is forwarded on 4 leading to *C*.
4. At *C*, the PDU is forwarded on 8, leading to *D*.
5. At *D*, the PDU has reached its destination.

These two examples show us:

1. Routing table sizes when addressing nodes are significantly smaller than routing table sizes when addressing PoAs.

2. A failing connections is fatal for a transmission when routing on PoAs, while routing on nodes allows adaptable re-routing in failure scenarios.

While these examples are minimal and simplified, these takeaways are true for real-world networks too. The former means that networks that route on hosts or processes have smaller routing table sizes, and the latter means that they adapt to network topology changes, which also has significance for how well they support mobility. Routing on PoAs seems to be the poor choice.

7.5 Evaluation of NDN

7.5.1 Critique of Premise

NDN's premise relies on the notions of *where* and *what*, which is problematic because they are both vague terms. As we discussed in Section 2.5, the location of an entity is another entity relative to which the entity can be reached. An entity that is the location of some other entity can itself be mobile: For example, a virtual host in today's cloud infrastructure may migrate from one machine to another, taking all its *whats*—its processes, endpoints and services—with it, despite the fact that 'a host' is typically understood to be the *where* of its *whats*. A virtual host is more a different, more complex, kind of *what* containing other *whats*, and associating it with a fixed *where* does not generally make sense. We argue that a whole architecture should not be based on these terms without them being very clearly defined.

In Zhang, Afanasyev, et al. 2014 it says: 'Distribution networks are more general than communication networks, and solving distribution problems via a point-to-point communication protocol is complex and error-prone'. This is unsubstantiated.

Jacobson et al. 2009, and many papers citing it, including Zhang, Afanasyev, et al. 2014, uncritically repeat the mantra that the Internet is a 'host-centric' architecture. However, the statement is vague: Yes, communication happens between hosts, at least if hosts are defined as PSs, however, communication also happens between processes, because the entities inside PSs that generate and consume communication will always be processes, and also between NICs, because all communication passes through a NIC before entering the network. The reason that NDN focuses on the 'the host' rather than the other entities might, we believe, be because of the mistaken and widespread belief that IP addresses identify hosts. As we have detailed in Chapter 4, an IP address identifies a PoA, not a host. This difference is crucial but it seems that it is not taken into account by many, including the authors of NDN.

Internet communication is not host-to-host, it is IPC: All transport-layer communication

happens from socket to socket, where each socket is bound to an AP; and all communication directly on IP is handled on a per-NIC, per-protocol basis, typically by an OS-process, or by a user AP binding itself to handle a specific protocol number for a specific IP address.

Although the exact difference between *where* and *what* is unspecified, and although the Internet is not based on a host-centric communication model but rather on IPC, Jacobson et al. 2009; Zhang, Afanasyev, et al. 2014 have a point, which boils down to the point that the IPC of the Internet is intrinsically tied to IP addresses, and that IP addresses are bound to PoAs and therefore to the network graph. We covered in Chapter 4 *why* the IPC of the Internet is bound to the network graph, and we also shown in Chapter 5 how an IPC can be made that is *not* tied to the network graph at all: It comes down to whether or not an architecture sensibly applies principles such as indirection, encapsulation, and locality to achieve modularity and dynamicity. In other words, we claim that the problems that NDN addresses have nothing to do with IPC (which the NDN community wrongly refer to as host-centric communication) but with modularity.

In summary, the premise that NDN presents seems to be: ‘The Internet is based on IPC, and its communication model is tied to the network graph, which leads to problems. Therefore, we conclude that IPC is the problem, and we should change it’ (Zhang, Estrin, et al. 2010). Besides this premise being logically faulty, it is not substantiated, and based on the analysis we have presented in Chapter 5 and Chapter 4, we believe it to be wrong.

Notes on What To Identify

In the Internet, PDUs are provided with an address that tells each relay in the network on which port to forward the PDU. In NDN, PDUs are provided with a name that tells each relay in the network on which port to forward the PDU. There is no conceptual difference. The difference lies in why, when, and how forwarding information is updated.

NDN (Zhang, Estrin, et al. 2010) claims that NDN names identify content, because the result of the chain of forwards will be that the PDU ends up by the content (if the PDU is an interest type) or by the entity that requested the content (if the PDU is a data type).

We propose the following alternative view: Labels identify the destination process of the PDU, i.e., the process that needs to react to it in some way. What ends up being the destination depends on the distributed forwarding state in the network.

7.5.2 Concerns of the Application in NDN

The network API task is similar both for NDN and traditional applications. They do differ depending on the network API, but not depending on whether or not the client application is an NDN or a traditional application.

The NDN protocol task needs to provide all the functionality specified for NDN. This includes building outgoing PDUs and parsing incoming PDUs which involves signing and securing them according to NDN specifications or best practices and creating valid NDN names and using them in agreement with other applications and forwarders. There are libraries, for example CCNx (CCNx 2016) or NDN-CCL (NDN-CCL 2016), that can help developers implement this task.

NDN explicitly does *not* handle functionality belonging to the transport layer in the Internet model, such as end-to-end flow control, reliability, ordering, and fragmentation. Such functionality is left to be handled by the application protocol task.

Transport mechanisms can be difficult to implement, and sometimes require intimate network knowledge. For example, because NDN does not support fragmentation, an NDN application reacting to an interest will have to find out the maximum transmission unit (MTU) for the expected path of its data PDU.

If the application protocol has a good fit with ICN semantics, it will require fewer steps to build based on a service using an ICN API than on a service using an IPC API, such as the socket or RINA APIs. However, NDN does not provide a service with such an API, so the point is moot.

In summary, a developer of a traditional application needs to handle the network API and its semantics, the application protocol and its semantics, and any data authentication, if needed. He or she can depend on the network service taking care of everything that has to do with transport.

A developer of an NDN application needs to know about all of the above except security of endpoints. Even though data authentication can be left to NDN libraries, he or she needs to know how to use the NDN data authentication and signature functions correctly and securely, so little is gained.

In addition, a NDN developer needs to handle flow control, reliability, ordering, and fragmentation, which is integrated into the Internet and in RINA.

7.5.3 NDN According to RINA

As we saw in Chapter 2, applications always use an application protocol that they both agree on. This allows them to exchange information that makes sense to each other and can affect each other's state and thereby behaviour.

The application protocol's only purpose is to exercise the states of the applications; it has nothing to do with facilitating the communication itself. In a black-box service architecture like RINA, everything about the communication between the applications is taken care of by the supporting stack, accessed via an abstract API. This situation is shown in Figure 7.17. Here, the applications have two tasks each:

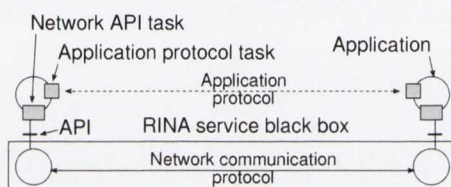


Figure 7.17: Inter-application communication in a black-box service architecture like RINA

1. An *application protocol task* for handling the application protocol.
2. A *network API task* to support the API that allows the application protocol task to send and receive data for the application protocol.

All the details of how to get data from one application to the other with some QoS guarantee is not the concern of the application developer who does not need to know *how* it happens; just that it *does* happen. If it does not, the application will be told about it. This *separation of concerns* is made possibly by having the communication service available as a 'black box'-like layer. It is what RINA seek to achieve.

In NDN, there are no APIs between the application and the NDN service. Instead, an application communicates with one or multiple nearby NDN forwarders using the NDN protocol directly. Forwarders communicate amongst themselves also using the NDN protocol. A first approximation to the situation in the NDN architecture is shown in Figure 7.18. However, the NDN protocol itself depends on one or more packet delivery services, such as Ethernet, IP, UDP, TCP, or possibly RINA. A more correct picture including this is shown in Figure 7.19. So, an NDN applications have three tasks each:

1. An *application protocol task* for handling the application protocol.
2. An *NDN protocol task* for managing the NDN protocol. The application protocol task

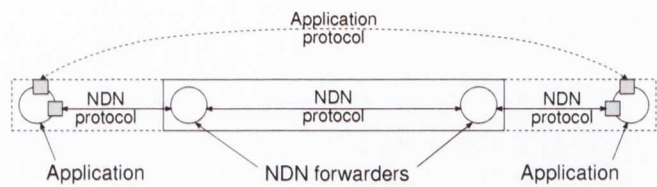


Figure 7.18: A first approximation to inter-application communication in the NDN architecture.

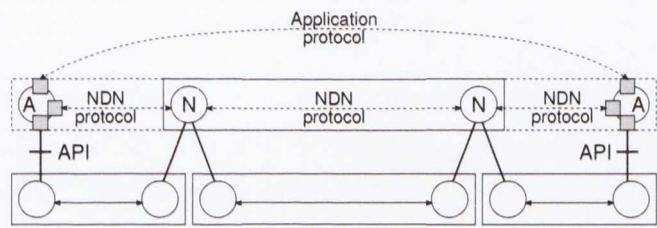


Figure 7.19: A better approximation to inter-application communication in the NDN architecture, now including supporting packet delivery service layers.

uses NDN to communicate, so it depends on this.

- 3. A network API task to support the API that allows the NDN protocol task to send and receive NDN PDUs.

Any protocol requires consensus amongst *everyone* participating in the protocol, and because applications and forwarders use the NDN protocol for communication, if something in the NDN protocol changes, both types of components need updating. So, a direct consequence of NDN applications communicating with forwarders by means of the NDN protocol rather than via an API is that updates to the protocol affects applications in NDN. This also means that any change in forwarders that will affect the NDN protocol, will affect applications too. The result is a high degree of coupling between applications and the network in the NDN architecture.

The coupling between applications and protocols have already caused the current Internet architecture to shy away from too extensive changes, despite the abstraction provided by the socket API. This has locked the Internet in a rigid state of stagnation, where no large changes can take place without insuperable consequences. In NDN, the coupling is even stronger, making the NDN architecture even more rigid.

7.5.4 Overloaded Pervasive Names

Zhang, Estrin, et al. 2010 states that ‘names are opaque to the network’. This is not correct. On the contrary, the structure of names in NDN is at the core of the architecture.

7.5.5 Names and Addresses

At the core of both the IP protocol and the NDN protocol is the longest-prefix matching of PDU labels with entries in routing tables. Routing table entries contain a label prefix and an outgoing port. When a PDU is forwarded, its label is matched with the entries in the routing table. The port(s) of the entry with which the label matches with the longest prefix will be the one the PDU is forwarded on. In IP, only one port per entry is allowed, while in NDN multiple ports are allowed.

The fact that routing entries map labels to physical PoA on which to forward PDUs makes both types of labels, whether or not they are IP addresses or NDN names, bound to locations in the network graph at any given point in time. It is just that IP addresses are typically bound to the same location longer than NDN names are.

So, the routing table mappings tie the PDU labels to the network graph, but applications want to use labels that are independent of the network graph.

For applications to use labels that are independent of the network graph, there are two options: Either, there should be a level of indirection between the labels that applications use and those used for forwarding, in which case the routing information can be kept mostly static; or the labels used by applications and for forwarding are statically bound (for example, identical), in which case the routing information needs to be dynamic.

NDN has chosen the latter.

The point is that the indirection in routing tables is a harder and more distributed problem than the indirection in higher layers, because forwarding needs to consult the routing table in each router and needs to do so at line speed. To change a location of a label either requires changing all routing tables in the network, or it requires providing a single mapping from the application name-space to the network one at the moment the PDU is sent from the application.

While DNS does provide a sort of mapping from application labels to network labels, it is incomplete and updates are several orders of magnitude slower than needed (hours rather than (milli)seconds).

7.5.6 Architectural Issues

Apart from the premise being questionable as argued in Section 7.5.1, the resulting architecture has at least two important issues:

1. NDN does not specify a clear system boundary, i.e. a boundary between the applications that are the users of the NDN system, and the NDN layer that provides the NDN service.
2. The many layers of semantics contained in names and the fact that names are shared between all components make the architecture rigid by upholding interdependencies between components.

7.5.7 Lack of System Boundary

Complex systems in most fields, including computer science, are typically designed using the abstraction of 'black boxes'. A black box is a system with well-defined interactions with other systems, while its internal behaviour is unspecified.

In Computer Science, a new system to be designed typically uses the black box abstraction as its point of departure to map out what the system shall do in relation to its surroundings. This involves specifying, at least abstractly, the interfaces of the system.

The design and implementation of the system seeks to specify the internals of this black box to comply with the interfaces in the best way possible according to some agreed parameters, which can include: Efficient use of resources (computing power, memory consumption, storage room, electricity, man-power, etc.), flexibility, ease of future expansions, or portability.

CCN and NDN does not explicitly define such a high-level system abstraction that is independent of its perceived and expected internals. In Jacobson et al. 2009, the service is defined bottom-up in terms of internal components and types of PDUs, their names, and how they should be handled. The architecture focuses on how to create, forward, and react to interest or data PDUs.

There are two APIs that do exist, CCNx (CCNx 2016), is a low-level one that defines operations such as listen, receive, and send and provides tools for applications to create and parse NDN packets and names used as inputs and outputs for these operations.

When applications construct PDUs, they need to adhere to data formats and semantics required by the network, as well as to the formats and semantics required by the applications they communicate with.

As a contrast, consider RINA, which makes a clear boundary between the part of the PDU that is used to support data transfer and the part that is not. The former is the PCI, the latter the SDU. Applications only ever see the SDU. Any intra-DIF information is contained in the PCI by the DIP assembling the PDU. While the DIP have access to the PDU, and can transform and read it (for example to support encryption, fragmentation, and data corruption checks) it never interprets the information it contains—i.e., it is not concerned about its semantics.

This separation provides isolation between applications and the DIF.

7.5.8 NDN Explained in RINA Terms

Some of the terms introduced in RINA transcends architectures. The concept of a distributed application facility (DAF), for example, is a distributed application that provides a service to its users via a well-defined interface. It is a ‘black box’ in the sense that its characteristics are based solely on its interface and how we expect it to behave.

A DIF is a DAF that provides a specific service, namely that of IPC. ‘PDU’ is a general term covering what has been referred to as packets, messages, or datagrams in other contexts.

We explain NDN using the terminology of RINA to point out some of the problems of NDN.

An NDN service can be abstracted as a DAF. It would expose its service to user applications via some API. A quite natural choice for the NDN DAF would be to have two parts to its NDN APIs: One for consumers and one for producers. Of course, an application might assume both roles and use both parts; that they are different parts means that one part does not require the other; that they can be used independently; that their states are not entangled.

As an example, consider a consumer API part consisting of a `get` operation to fetch a data object based on its name:

$$status, data \leftarrow get(name)$$

and a producer part consisting of a `put` operation to make a data object available to the NDN service, and an operation to cancel it:

$$status \leftarrow put(name, data)$$

$$status \leftarrow cancel(name)$$

The `get` operation returns a status code and the desired data object, if it is found and no

errors have occurred. The put operation would block until a cancel with a matching name is invoked. While blocked, the specified data would be available under the given name to the NDN service. If *data* is allowed to be a data-generating function, data can also be generated dynamically.

Such a service abstraction would allow the NDN DAF to only share with its user applications the information that they require and nothing more: Names and data.

The NDN DAF could be implemented using whatever means necessary, perhaps using the protocols and PDUs types specified by NDN, if they can efficiently support the desired service. Because of the well-defined and encapsulating API, any choice in the design and implementation of the NDN DAF that does not affect the API or its input or output would be hidden from user applications.

However, NDN does not define such a service abstraction. Rather, we can think of the NDN architecture as an NDN forwarding DAF without a north-facing service API, where the interaction with its user applications happens via the same methods as the internal DAF communication, i.e. through lower-level DIFs using IPC.

This raw interface does not provide encapsulation. NDN applications create NDN PDUs and send them to the nearest NDN forwarder via IPC.

7.5.9 Premise of NDN

1. According to Zhang, Afanasyev, et al. 2014, our global network should be a 'distribution network' providing data dissemination rather than a 'communication network', providing IPC (our interpretation, because Zhang, Afanasyev, et al. do not define what they mean with 'distribution network' or 'communication network'). Zhang, Afanasyev, et al. claim that 'Distribution networks are more general than communication networks, and solving distribution problems via a point-to-point communication protocol is complex and error-prone'. Despite that this claim is not substantiated, it serves as the motivation for the whole NDN research project (Zhang, Estrin, et al. 2010).
2. Changing the main abstraction from IPC to data dissemination can solve some of the problems of the current Internet, such as availability, mobility, authentication, and scalability.

The argument for Item 1 is that data dissemination is an abstraction used by the majority of networked applications, and therefore it should also be the main service provided by a global network.

That argument is faulty for two reasons: First, what is popular might change over time. If architectures should cater to the whims of applications, we could be changing architectures forever. Second, there can be a number of applications not using the popular abstraction that might be left with a ill-suited abstraction. These applications, despite not being popular, might still be important.

We argue that the core abstraction provided globally should rather be based on universality. In other words, we should choose the abstraction with which general distributed applications and services can be built, rather than one that only can support a certain popular type of application, leaving other types of applications with an unusable or, at best, cumbersome abstraction. As RINA has shown, a general DAF relies on a DIF that can provide an IPC service. Thus, IPC is such a universal abstraction.

The question is: Is NDN's data dissemination model also such a universal abstraction? The answer seems to be 'no, not quite': For example, to create bi-directional streams for two programs to communicate and provide a sample telephony service, Mosko overloads NDN name semantics originally intended to identify pieces of data only to also contain return addresses and sequence numbers. This turns NDN names into a combination of name and data: interest PDUs are routes based on the first part of the name, while the last part of the name defines the action expected of the recipient of the interest. Overloading the name semantics in NDN is a prevalent practice that can contribute to complexity.

Data Dissemination Abstraction Solves Internet Issues

NDN claims (Zhang, Estrin, et al. 2010) that many of the problems of the Internet are caused by the mismatch of abstractions between those used in applications and those provided by the network. We have seen that the problems with TCP/IP are *not* caused by a mismatch in abstractions, but rather by ignoring principles that leads to simplicity and modularity, such as encapsulation, indirection, and locality.

Specifically, NDN claims to solve issues of scalability, mobility, and security (Zhang, Afanasyev, et al. 2014; Zhang, Estrin, et al. 2010).

7.5.10 Conclusions

NDN has defined a very simple protocol at the core of its architecture based on the premise that this would keep the architecture as a whole simple too. We have seen that this is not the case. For a certain level of quality, certain mechanisms need to be in place. If they are not part of the network layers, they will need to be somewhere else. The result of the

simplicity of NDN’s data plane is that the complexity has been pushed to applications, to name semantics, and to the control plane.

7.6 Properties

7.6.1 Flexibility

Table 7.6 shows which changes are possible at run time for each architecture. Changes not possible at run time require recompilation, relinking, or a change to an application’s source code.

Possible at run time	Architecture		
	TCP/IP	RINA	NDN
Changing QoS	(✓)	✓	
Integrating new layers		✓	
Changing address format		✓	
NRS change		✓	N/A

Table 7.5: Run Time Flexibility

The QoSs and semantics provided by TCP/IP and NDN are hard coded in the protocols, and the choice of protocol, and thereby QoS is chosen by applications at run time. However, the application can only choose between the QoS of the protocols provided by the architecture. In TCP/IP, QoS is changeable at run time by specifying a specific available transport protocol. One can only choose transport protocols supported by both ends as well as by intermediary firewalls and middleboxes, which in practice limits the choice to TCP, UDP or (in some cases) SCTP. Therefore, we assign a partial ‘(✓)’. In NDN there is no choice, because there is only one protocol with one QoS. In RINA, the QoS is widely configurable per flow by the application at runtime, and is not limited to only a few choices.

In TCP/IP and NDN layers are designed to be different in scope, routing, addressing, protocols, and other aspects. The way they integrate is not specified abstractly, so it is implementation-specific and hard coded. Combining existing layers in new ways or introducing new layers require specialised implementations for each new combination. In RINA, scope, routing, addressing, and many other aspects are configurable. The protocol is fixed to EFCP, but it is configurable. Layers can be combined using the same API as that used by applications.

In TCP/IP, the EID format, which includes the address format, is hard coded in the API, i.e., the specific address format is integrated into the API. Any change requires application source adaption. In NDN, the low-level APIs expose all protocol syntax and semantics,

including addresses, to the application. Any change in either require application source adaption. In RINA, address formats are configurable per layer. The per-layer encapsulation of addresses prevents applications from knowing address formats, which can therefore be changed while applications are running.

In TCP/IP, applications manage name resolution, specifically its integration with the application itself and the socket API. When resolving a host name, applications must chose and explicitly call a specific name resolving server, and the returned result is specific to that service, e.g., calling a specific DNS server at a specific IPv4 address and getting back an IPv4 address. Any change to name resolution mechanisms, syntax, or semantics that affects the API would require application source changes. In NDN, there is no NRS because the identifiers used for routing are also used for names. In RINA, name resolution is encapsulated in each layer; thus each layer can freely choose its own NRS without applications being affected.

Table 7.5 shows that RINA is highly flexible, while TCP/IP and NDN is not. The reason is simple: RINA has a clearly defined, abstract API that is specified independently of any implementation and choice of addressing or NRS. NDN lacks an abstract API altogether, and instead relies on specific implementations (e.g., NDN-CCL or CCNx) where individual PDUs are constructed and interpreted directly by the applications. TCP/IP has a de-facto standard API, but it does not abstract the NRS, and it exposes addresses, which can then not be changed without altering the API.

The key is top-down design: The APIs should come first, the implementation second. That way, the APIs will be implementation-independent, which allows implementation changes without API changes—which is what provides flexibility. Doing it the other way around leads to inflexible systems; implementation details, such as the ones, we have seen here: Addresses or protocol formats appear in APIs.

Another key component is encapsulation with which implementation specific data structures used inside components (such as addresses) can be hidden and mapped to more abstract ones in the APIs (such as names) using indirections. Encapsulating functionality (such as name resolution) prevents future changes to it having widespread consequences.

7.6.2 Mobility

In TCP/IP, the flow between two applications is conflated with the connection between two sockets; there is no distinction between the two. Connections and sockets are identified via their EIDs, which has the IP address hard coded into it. The IP is bound to the PoA in

IA	Flexibility
TCP/IP	-1
Modern	-1
NDN	-2
RINA	2

Table 7.6: Flexibility Assessment

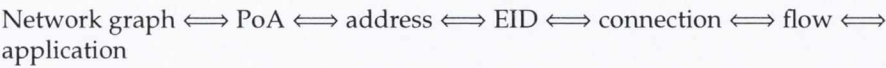
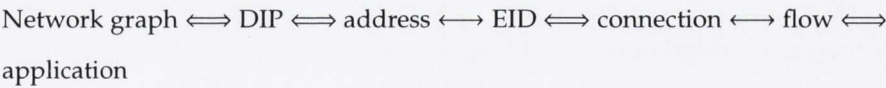


Figure 7.20: Bindings and Indirection in TCP/IP.

routing tables and in the OS. If $\longleftrightarrow$ signifies a static binding and $\longleftrightarrow$ a dynamic indirection, the situation is as shown in Figure 7.20. There is no indirection, a change of PoA breaks the flow and requires the application to handle the recovery.

Even the proposal for Multipath TCP (Ford et al. 2013) suffers from the lack of indirection and thus has the same problem.

In RINA, connections and flows are different. A flow is facilitated by one or more connections and applications do not manage connections; they manage flows. If a connection breaks, the flow still persists. Furthermore, EIDs are not bound to addresses. This means that even if an address changes, EIDs are not invalidated and connections persist. In addition, addresses are bound to DIPs rather than PoAs, and the PoA is just a part of the transitory network path, which may change at any moment. The situation is as follows:



Two levels of indirections exist: Between addresses and EIDs, and between connections and flows. Applications handle recovery of flows, but as explained below, flows rarely break. If addresses change, the connection can still persist, if it is remapped to the new address. If the connection breaks anyway, because remapping is not possible, or for other reasons, the flow still persists. Connection error recovery is handled in the network, transparently to the application. Only if the flow breaks, does the application need to react. This may happen, if the other application loses connectivity altogether. These two levels of indirection exist in each layer: The DIP of an upper layer is really an application of a lower layer.

Many modern Internet augmentations attempt to introduce indirections at different points between the PoAs and the application while keeping existing invariants as constant as possible. SDN makes it easier to manage routing information and connection paths by

controlling the binding between addresses (or EIDs) and PoAs. MIP and loc/id introduces a new level of identifiers so that PoAs may change without EIDs changing.

IPv6 does not change any of these bindings, and so it does little for mobility.

For NDN, the picture is different for two reasons: Firstly, NDN is asymmetric because the interest’s path from client to server is based on long-term routing state, while the data PDU’s path from server to client is based on short-term state induced by the breadcrumb trail left behind by the interest. Secondly, although endpoint processes do exist, the model does not assign identifiers to them specifically—except if an NDN name coincidentally identifies one.

The path from the server to the client based on the breadcrumb of the interest is transient, so we focus on the path from the client to the server, which looks as follows:

$$\text{Network graph} \iff \text{Name} \iff \text{object}$$

There is no room for indirections: The name is tied to the object using cryptography, and the network routes on names directly. Moving objects between servers requires changes in the routing state reflecting the network graph.

Using SNAMP (Afanasyev, Yi, et al. 2015), a loc/id indirection is introduced:

$$\text{Network graph} \iff \text{Locator name} \longleftrightarrow \text{Identifier name} \iff \text{object}$$

This suffers from the same weaknesses as any other loc/id solution

Mobility is a question of bindings and indirections. If there are no indirections, bindings are static, and things cannot change association/location in relation to each other. With indirections, bindings can be made dynamic and things can move. TCP/IP’s lack of mobility is due to its static bindings all the way from the network graph to the application. NDN also lacks dynamic bindings, so objects and the servers providing them cannot move without interrupting service; however, the short-lived state of data PDU routing provides for client mobility even though the state is technically just as rigid. RINA provides several indirections at key points and has explicit support for mobility.

IA	Mobility
TCP/IP	0
Modern	1
NDN	0
RINA	2

Table 7.7: Mobility Assessment

7.6.3 Scalability

TCP/IP’s scalability relies on aggregation, which relies on IP addresses being aligned with the network graph. Network-independent IP addresses enabled with SDN, MIP and loc/id pushes towards a less scalable architecture. Renumbering, which is required to keep the IP address space aligned to the network graph, is avoided because of the flexibility issues mentioned in Section 7.6.1. IPv6 also relies on aggregation. Due to its larger address space, the scalability issues are potentially more serious.

Scalability in RINA relies on the scoping and hierarchy of layers, as well as on each individual layer’s scalability. Upper layers with many nodes can apply aggregation similar to that of IP, or utilize other scalable routing mechanisms such as hyperbolic routing (Boguñá et al. 2010). Several layers may also coordinate their routing mechanisms to provide scalability, as suggested in Day, Trouva, et al. 2011. The flexibility of each layer to choose its own routing method and addressing format allows for both of these solutions.

NDN scalability relies on aggregation as well, but unlike in TCP/IP where addresses can be assigned based on network location, NDN names are designed to be independent of the network, which makes aggregation impossible. The fundamental NDN approach is therefore not scalable. SNAMP (Afanasyev, Yi, et al. 2015) introduces a loc/id-like solution for this reason, but suffer similar problems as the loc/id solutions for TCP/IP (Day 2008b; David Meyer and Lewis 2009).

Based on these observations, we assign the scores as shown in Table 7.8.

IA	Scalability
TCP/IP	0
Modern	-1
NDN	-2
RINA	2

Table 7.8: Scalability Assessment.

7.6.4 Security

TCP/IP has no built-in security. There is no confidentiality mechanism: Everything is cleartext. Checksums can detect accidental changes, but deliberate mangling by adversaries is neither prevented nor detected. There is no authentication to join a network, so anyone can interact with everyone else. Well-known protocol and port numbers make it simple to define targets. There is no provenance and addresses can be spoofed. Availability guarantees are poor: Applications can become unavailable, if they or their host are configured incorrectly, if

targeted in a DoS attack, if not properly programmed to handle network changes such as connection loss or address changes, or for a variety of other reasons.

Modern augmentations have improved this picture. IPsec, for example, adds integrity, confidentiality, authentication, and provenance. However, basic TCP/IP is still prevalent, and IPsec is not deployed except where there are explicit needs, for example, for VPNs. DoS attack vectors are still the same.

NDN provides many security features, for example, integrity, authentication, and provenance via PDU signing and certificates. Confidentiality is up to the applications. While most literature agrees that NDN is better secured against DoS attacks than TCP/IP due to the pull-semantics nature and security features of NDN, new kinds of DoS attacks are possible (Afanasyev, Mahadevan, et al. 2013).

RINA manages authorisation and scoping layers. Any DIP joining a layer is authorised before it is allowed to send and receive data, and target addresses are not well-known and difficult to guess, both of which helps to prevent DoS attacks. Integrity, provenance, and confidentiality can be configured per layer. The network manages recovery after address changes and connection loss and ensures correct configuration, which together with the reduced DoS attack vector improves availability. authorisation between APs using a layer happens between those APs, so it happens outside of RINA—however, an AP running on an unauthorised node will not be able to even communicate on the layer, because the node cannot join the layer without being authorised when authorisation is required by the layer.

Based on these observations, we score each IA as shown in Table 7.9.

IA	Security
TCP/IP	−2
Modern	0
NDN	1
RINA	2

Table 7.9: Security Assessment

7.6.5 Simplicity

We assess the simplicity of the IAs based on 1) functionality, and 2) the number of protocols and message types in each IA.

7.6.5.1 Functionality in IA and Applications

If a certain service is required for an application’s task, and only a part of the service is provided by an IA, the application will have to provide the missing parts itself. This means that an IA that fails to provide a function that most applications need, in effect, pushes the function implementation to the applications. While such an IA is in itself simpler than a more comprehensive IA that provides more functions, the complete combined system consisting of both the IA and its applications will be more complex, if what could be shared functionality is implemented individually in most applications. On the other hand, an IA that provides a function that only a few applications use is unnecessarily complex. A sensible IA will make a balanced choice as to the functions it should or should not provide.

The IAs, we analyse in this dissertation, make very different choices as to which functions they provide and which they do not. This we summarise in Table 7.10. Functions provided by each IA adds to their complexity, but eliminate any complexities in user applications or auxiliary programs necessary to achieve the same service.

Provided functionality	Architecture			
	TCP/IP	Modern	RINA	NDN
IPC service	✓	✓	✓	
CDN service				✓
Transport and bookkeeping	✓	✓	✓	
Mobility		(✓)	✓	(✓)
Name resolution			✓	
Connection management and recovery			✓	
Layer instantiation and customisation			✓	
PDU construction	✓	✓	✓	
PDU interpretation	✓	✓	✓	

Table 7.10: Functions provided or not provided by various IAs.

TCP/IP provides an IPC service with transport functionality, including sequencing, reliability, multiplexing, etc., so applications do not have to handle this themselves. Mobility is not provided, and applications that need mobility will need to exploit name resolution and connection management (and recovery) to provide this themselves where possible. Name resolution and connection management are not provided by TCP/IP, so they are also left up to each application. In TCP/IP the number, types and order layers are fixed—if new layers are needed, it requires special customisation in the network and in the host OS’es. For the layers available, the QoS is fixed and implemented via specific protocols. A CDN service is not provided, so content dissemination is also an application matter. The specialised layers and the functions provided make the architecture somewhat complex. Applications can rely

on the functions it does provide, of which transport is probably the most complicated, but have to take care of mobility themselves.

Modern TCP/IP augmentations add some support for mobility via SDN and various loc/id solutions that allows applications to keep their IP addresses despite moving, but solutions vary, and applications cannot in general rely on mobility support everywhere, so they will still need to handle mobility themselves to some degree. The augmentations make the IA more complex and fragmented than pure TCP/IP. Applications cannot rely on new functionality to be ubiquitous, so they still have to be able to fall back on pure TCP/IP, which makes at least as complex as applications running on classic TCP/IP.

Like TCP/IP, RINA provides an IPC service with transport functionality, but in addition it handles mobility, name resolution and connection management as well. The accompanying complexity is therefore not required in each application, but can be abstracted once and for all by RINA. In RINA, layer configuration and the QoS for each flow can be customised at run time. CDN services are not provided, but the object-centric CDAP has more features than a mere transport protocol and thus can support the building of CDN services. While each layer provides a lot of functionality, it is not always at the expense of additional complexity. Generalising the transport protocol, for example, reduces the number of protocols, while providing added functionality and flexibility in choosing QoS. The recursion of layers further reduces the complexity considerably. Applications can be quite simple due to much functionality being provided by RINA.

NDN provides what is basically a CDN service, where clients can be mobile but servers cannot. Its APIs are low-level and implementation-bound, and applications build, interpret, send and receive individual NDN PDUs via libraries. Transport functionality is not provided or abstracted away, so applications will have to handle sequencing, retransmission, and all bookkeeping, for which libraries are provided. Name resolution or any general indirection between names used in applications and the names used for routing have to be managed by the application, including using SNAMP (Afanasyev, Yi, et al. 2015). Connections do not exist in NDN, so any state synchronisation and recovery of errors is left to applications too. If an IPC service is needed, it has to be built using the CDN-centric NDN primitives. This is non-trivial, as demonstrated in Mosko 2014, and is once again left to the applications. While NDN itself is simple, it provides only few functions generally required by distributed applications, which then become unnecessarily complex.

Our simplicity assessment based on functionality is summarised in Table 7.11.

IA	Simplicity	
	Apps	IA
TCP/IP	1	-1
Modern	1	-2
NDN	-2	1
RINA	2	1

Table 7.11: Simplicity assessment based on functionality.

7.6.5.2 Protocols and Message Types

Complementary to the previous section’s judgement-based simplicity-measure, we here provide a quantitative one based on two measures: 1) the number of protocols, and 2) the number of message formats. We only include protocols necessary for actual data communication over commonplace wired electrical networks, which means package-delivery protocols and transport protocols. We exclude other protocols, including application, routing, and name resolution protocols, as well as protocols for communication over the air or fibre.

The picture looks as shown in Table 7.12. For TCP/IP, we include one L2 protocol, namely Ethernet, which provides packet-delivery on which IPv4 in L3 may run. In L4, we include TCP and UDP, each of which provides a different QoS. Each of these protocols has a single message format.

NDN needs to run on a package-delivery protocol to be able to send and receive packages. It can run on Ethernet, IPv4 or UDP. We have included Ethernet here as a minimum. The NDN protocol provides end-to-end data transfer, and has two message formats: Interests and data messages. For the modern Internet, the picture for data transfer is similar.

RINA’s EFCP provides both end-to-end data transfer, and can also work directly on the physical media as a substitute for Ethernet. It will layer several instances of EFCP to provide end-to-end data transfer. EFCP has two message formats: DTP for messages carrying data payload, and DTCP for control, flow and management messages without payloads.

Summarising, we see that TCP/IP has 4 protocols and a total of 4 message types; NDN has 2 protocols and 3 message types, and RINA as 1 protocol and 2 message types. If number of protocols and number of message types is an indication of IA complexity, these metrics support our judgement-based scores in the previous section. We, however, do not use the metric to score the IA, but we leave it as a supplement.

IA	Protocols	Message Formats
TCP/IP	TCP	1
	UDP	1
	IPv4	1
	Ethernet	1
	4	4
NDN	NDN	2
	Ethernet	1
	2	3
RINA	EFCP	2
	1	2

Table 7.12: Protocols and number of message formats required for end-to-end data transfer.

7.6.5.3 Summary

We argue that the simplicity of an IA need to reflect the combined simplicity of the IA itself as well as that of its applications. We choose to give twice the weight to application simplicity. The number of protocols and number of message formats.

Our assessment is summarised in Table 7.13.

IA	Simplicity		
	Apps Weight: 2	IA Weight: 1	Combined Weighted avg. (rounded)
TCP/IP	1	-1	0
Modern	1	-2	0
NDN	-2	1	-1
RINA	2	1	2

Table 7.13: Simplicity Assessment. The combined assessment is a weighted average, rounded to nearest integer, with application simplicity weighing twice as much as IA simplicity.

7.7 Principles

In this section, we discuss the principles presented earlier and discuss them in the light of each of the systems considered.

7.7.1 Indirection

In this subsection, we discuss the principle of indirection: First, we discuss systems that do not apply the principle in specific cases. Second, we discuss systems that do apply indirection in specific cases.

7.7.1.1 System Cases Not Applying Indirection

NDN The lack of indirection between the application namespace and the routing namespace allows for a simple architecture without the need to support a mapping function. The cost is reduced flexibility, because applications and routers are coupled in choice of names, which are then difficult to ever change; reduced scalability and mobility because unrestricted location changes of producers require global routing table updates or loc/id-like solutions and complex applications because they have the responsibility to choose content names, but this choice greatly affects network performance and needs to be based on intricate knowledge of the network graph and expected network dynamics.

TCP/IP In TCP/IP, there is no indirection between addresses, CIDs, flow identifiers (FIDs): The first is included in the two latter, which are identical and identifies the socket. This couples applications, protocols, and routing mechanisms, which are all locked in their choice of identifiers. The consequence is reduced flexibility, because routing or protocol implementation changes are seriously restricted, lest all applications break; reduced mobility, because address changes invalidates CIDs and FID requiring applications to recover; and reduced security, because CIDs and FIDs may be guessed because of their limited range and open up attack vectors. In the end, scalability suffers too, because it depends on the ability to aggregate addresses, which in turn requires occasional renumbering of addresses in order to keep them aligned with the network graph, which due to connection invalidations and recovery being the responsibility of the applications is mostly avoided.

7.7.1.2 System Cases Applying Indirection

TCP/IP TCP/IP applies a layer of indirection between URLs and FEIDs via DNS. This allows applications to recover from broken connections due to changed addresses, where the FEID has changed but the URL (or similar peer identifier) has not, allowing for mobility assuming that DNS can provide up-to-date mappings. The DNS redirection also provides a limited degree of flexibility in address formats. In particular, an application using the `getaddrinfo` POSIX API (Gilligan et al. 1999) for name resolution can be agnostic to whether

or not the address contained in the FEID is in IPv4 or IPv6 format. However, applications can never handle an all new address or FEID format without code changes, recompiling and/or relinking.

RINA RINA has several indirections in each layer: There is an indirection between FIDs and CIDs and vice versa, between FIDs and address, and between address and PoAs, each of which takes the form of a PID, referring to a FID in a lower layer, in which the same type of mappings repeat. The indirection between the FID and the addresses is provided by the RIB, which conceptually constitutes a layer-scoped distributed NRS. This allows addresses to change without altering the FID, shielding applications from knowing (and caring) about it. The indirection between a FID and its CIDs is local and provided by the FAI, one of which exist per flow endpoint. This allows connections to terminate and spawn independently from the application. The indirection between addresses and PoAs is provided by the FIB, equivalent to a routing table and typically updated using a layer-wide distributed routing algorithm. In unison, these indirections provide multihoming, mobility, and flexibility: Multihoming because one FID can map to several CIDs (allowing several connections per flow); mobility because this mapping can change over time, and because one address can map to several PoAs (allowing connections with dynamic paths); and flexibility because the address format and routing algorithm are kept internal to the layer and never leaked to the application, which only ever see FIDs in the form of ANs or PIDs.

7.7.1.3 Summary

Where indirection is applied, it provides primarily mobility and flexibility.

It provides mobility because identifiers on one side of the mapping may change without affecting the other side of the mapping: The dynamics of one side is hidden from the other side by an equally dynamic mapping. If the dynamics of one side are caused by location changes, these can be made invisible to the other side. The point of mobility is that changing location happens transparently and without involvement from entities that does not need to know about it.

Indirection provides flexibility because the format of the identifiers of one side of the mapping can change without changing that of the other side, although the mapping itself will need updating. The mapping thereby decouples the two sides, providing modularity and independence. An early example of such decoupling using indirection can be found in the Emerald system (Black et al. 1986; Jul et al. 1988) where an object is referenced using an

object reference that provides an indirect reference to the IP address of the location of the object. This allowed object to freely move around the system completely transparent to any application referencing them.

Secondarily, indirection provides security and scalability. It provides security because indirections limits the scope of identifiers, outside of which they become useless, including for attacks. It provides scalability because of the flexibility: Adjusting identifier formats to match scalability requirements within a limited scope and period of time becomes possible.

7.7.2 Encapsulation

In this subsection, we discuss the principle of encapsulation: First, we discuss systems that do not apply the principle in specific cases. Second, we discuss systems that do apply the principle in specific cases.

7.7.2.1 Cases without Encapsulation

NDN In NDN, there is one L3 layer with global scope and unrestricted access. This opens up for anyone to try to attack anyone else in the network. The network state might restrict interest PDUs with certain names from reaching certain parts of the network, but no predetermined scoping, access control or other mechanisms exist to limit the number of adversaries that may attempt attacks in the first place.

TCP/IP Similarly, in TCP/IP, there is fundamentally one L3/L4 layer with global scope without access control, where anyone can attack anyone. Through the use of NAT, firewalls, and VPNs, scoping has in the modern Internet been applied as an add-on to the basic model, limiting attack vectors.

IP addresses and CEIDs are not encapsulated inside the layer, but exposed to applications via the APIs. This affects mobility and flexibility, as discussed earlier.

Name resolution, for which DNS is used, is not encapsulated but is invoked explicitly by applications before they interact with the TCP/IP service via the POSIX API. While name resolution in itself provides an indirection enabling mobility and some flexibility, the fact that it is not encapsulated in the service makes changes to it difficult because it would affect all applications. This makes for a less flexible architecture. This affects mobility indirectly because changes that would improve mobility are also hard to implement.

7.7.2.2 Cases Applying Encapsulation

RINA In RINA, each layer has a well-defined scope and access is limited to authenticated DIPs. Unauthorised DIPs cannot send or receive PDUs. The limited scope and access control limits the attacks surface by preventing unauthorised entities from sending and receiving PDUs. This increases security. Scoping can help to improve scalability by keeping the effects of changes local.

Name resolution is encapsulated in each layer. Changes to each layer's NRS is therefore invisible to applications, which means that it can be updated without breaking applications, as long as the layer adapts accordingly. Furthermore, RINA encapsulates connection management and any handling of addresses. This means that changes to protocol formats, address formats and ultimately routing mechanisms only affects the layer, which can adapt accordingly and shield the effects from applications. This provides flexibility to allow layers to adapt to new solutions. The encapsulated connections management means that the layer handles error conditions which can be done in a consistent way. This improves mobility.

7.7.2.3 Summary

Encapsulation means to limit the scope of a system or certain aspects of a system. This can be used to hide implementation details from users, allowing these details to change without affecting users, increasing the system's flexibility. It can also be used to restrict access, providing added security, and it can help to confine effects of changes or updates to prevent them from propagating globally, improving scalability and mobility.

In conclusion, when applied to IAs as exemplified in the previous sections, encapsulation leads to greater flexibility, security, mobility, and scalability.

7.7.3 Extensionality

Extensionality means to define a system from its extensional properties, i.e., *what* it does and how it is or should be used by other systems, rather than from its intensional properties, i.e., *how* it does what it does, how it is implemented.

First, we discuss systems that do not apply the principle in specific cases. Second, we discuss systems that do apply the principle in specific cases.

7.7.3.1 Cases without Applying Extensionality

TCP/IP The socket is the external hook used by users of the socket API. The choice to make it equivalent to the PM of a transport protocol connection breaks its extensionality

and makes its behaviour depend on intensional properties, namely those of the transport protocols. This also conflates flows with connections in TCP/IP so that flows *are* connections. The result is reduced flexibility, because transport protocols cannot change without at the same time changing the exposed behaviour of sockets; and reduced mobility, because when connections break due to changed addresses (caused by the lack of indirection between addresses and CEIDs discussed in Section 7.7.1), flows break, forcing applications to recover, which they are not always equipped or designed to do properly.

An advantage of directly exposing the PM as the socket via the API is that there is no translation of specialised intensional behaviour of the protocol to a more general extensional behaviour of an abstract API; this makes the architecture simpler.

NDN NDN lacks a well-defined, extensional interface. This makes the exact service it purportedly provides hard to define. Instead, its service is defined intensionally and implicitly from its protocols, specifically from its two types of PDUs: Interests and data. These are based on content-dissemination semantics described loosely in Jacobson et al. 2009; Zhang, Afanasyev, et al. 2014, but never concretised.

The lack of an extensional API, independent of implementation and protocols, means that the APIs that have sprung up (e.g., CCNx and NDN-CCL) are closely tied to a specific protocol implementation. This makes NDN very inflexible: Even the slightest change in either protocol data format or semantics directly affects the APIs and all applications dependent thereof.

NDN also lacks connections and flows. This is because (even without a well-defined service specification) it defines itself as a content-dissemination network where long-lived state per definition does not exist. When applications find that they, after all, do need state for bookkeeping of large transfers, they need to maintain this state themselves. In the end, the lack of connections and other transport services in NDN makes NDN simpler, but moves the complexity to applications when such transport services are needed.

7.7.3.2 Cases Applying Extensionality

RINA RINA defines its service as being IPC and from that specifies an API. Its implementation is build around facilitating the API and thereby the service. The result is a completely extensional service defined by its interfaces, without assumptions as to how the service is implemented. Because the API do not expose intensional properties, the implementation can vary and the architecture become flexible in the many ways we have already outlined:

Routing, addressing, name resolution, etc., can all be changed without affecting the API and the applications that use it.

7.7.3.3 Summary

Extensionality provides flexibility because dependencies on intensional properties—e.g., implementation-specific properties—are eliminated. The drawback is added complexity because intensional properties have to be actively hidden and/or mapped to extensional properties.

7.7.4 Generality

In this subsection, we discuss the principle of Generality: First, we discuss systems that do not apply the principle in specific cases. Second, we discuss systems that do apply the principle in specific cases.

7.7.4.1 Cases without Applying Generality

NDN and TCP/IP Both NDN and TCP/IP have fixed, specialised protocols that each provides a specific QoS, such as unreliable datagram transfer, that cannot be adjusted. In TCP/IP, the various protocols have different APIs, so applications depend on the specific choice of protocol. In NDN, the dependency between applications and protocol is even tighter, because applications are expected to directly construct and parse the PDUs. This fixed QoS and interdependencies between applications and protocols makes the IAs inflexible.

Both IAs also have fixed, specialised layers, meaning that each layer is specialised to provide a fixed service and that the order of layers is also, in principle, fixed. In practice, the ordering is not fixed, and layers repeat. For example, TCP/IP backbones employ tens of stacked layers, including Generic Routing Encapsulation (GRE), MPLS, Ethernet, IPv4, IPv6, and UDP in various orders. Every time two layers need to interrelate, a mapping of addresses has to take place; in TCP/IP, for example, the mapping between IP addresses and Ethernet MAC addresses is facilitated by ARP. This mismatch between what is evidently needed in practice and what the model prescribes complicates network implementations and shows the rigidity of the architecture. In NDN, interoperation between layers is specialised as well. In particular, CCNx can only run on UDP, while NDN-CCL may run on both UDP and TCP, in both cases because those are the interoperation implementations that are provided. New ones are possible, but they need to be specifically engineered.

7.7.4.2 Cases Applying Generality

RINA RINA have a single, generalised transport protocol with configurable QoS. It provides a single, unified, generic layer that provides IPC, that can be configured and instantiated with specific QoS parameters so that e.g. routing, addressing, and security properties can be adapted to the environment and requirements of the individual layer instance. Layers can be freely combined via standard APIs.

The generality of each layer and the possibility to combine layer instances makes RINA extremely flexible. While its configurability, being a product of a high level of generalisation, makes for a somewhat complicated basic layer design, the fact that network stacks are built recursively with instances of a single basic layer type using its standard APIs makes RINA overall simple.

7.7.4.3 Summary

Generality in layer and protocol design provides flexibility through configurability and reduces the number of disparate layer types and protocols needed at the cost of making the general layer and general protocol that are provided more complex. However, the overall architecture can be made simpler by reusing a single complex generic layer and protocol instead of using different specialised but simpler layers and protocols.

7.8 Summary and Perspective

We have looked at four architectures: TCP/IP, its modern variant with augmentations, NDN and RINA; how they employ or do not employ four design principles: Indirection, encapsulation, extensionality, and generality; and how, for each architecture, this affects five properties: Flexibility, mobility, scalability, security and simplicity.

7.8.1 Indirection

We have discussed the indirections between identifiers at various levels in each architecture.

In NDN, we have seen that without indirections between routing identifiers and object names, the only two ways to support mobility is to update the mapping in routers between routing identifiers and forwarding ports, or to change the names used in applications as objects move. The former scales badly, the latter is not mobility, or at least requires a higher-level name system similar to DNS, defeating one of the purposes with NDN in the first place.

In TCP/IP, there is no indirection between the IP addresses in routers and the FEIDs used by applications, leading to the same problem as in NDN: Either IP addresses are constant during mobility—this requires routing tables to change to update routes, violates aggregation, and does not scale—or the FEID needs to change. The latter is facilitated with DNS, which requires applications to a) maintain a higher-level namespace consisting of URLs or FQDN and well-known ports, which can be mapped to FEIDs, and b) manage connections including determining when active FEID change or mappings in DNS are updated. The latter is how mobility is supported in TCP/IP.

In the modern Internet, the indirection that maps IP addresses to routes is made more dynamic or manageable with MIP or SDN, or new indirections are inserted with the various loc/id solutions that map IP addresses to CEIDs, thereby decoupling the two. The former have scalability issues caused by violating aggregation, which can be mitigated by combining it with the use of DNS and limiting the scope of routing changes. The latter requires a new NRS or extending DNS to provide the new indirection mapping and furthermore has its own problems (Day 2008b; David Meyer and Lewis 2009).

RINA have two indirections in each layer: A layer-wide directory at the top of the layer, mapping FEIDs to addresses, and a local FIB at the bottom of the layer, mapping addresses to outgoing ports. FEIDs need never change, and mobility is supported by updating the FEID-to-address directory. CEIDs are decoupled from FEID to allow multihoming, meaning that a single flow is supported by one or more connections, which can change over the lifetime of the flow. Because directories and addresses are layer-local, and because the user of the directory is the layer itself, addresses can be renumbered and address formats are customisable, both of which help ensue scalability.

In the examples we have looked at, mobility is supported where the principle of indirections has been applied and mobility is lacking where no or few indirections are employed. We believe this to be straightforwardly logical: Indirections decouple two kinds of identifier spaces from each other, allowing one to change independently from the other by continually rebinding the mappings between them. This dynamic mapping is what provides the degree of freedom to facilitate mobility.

7.8.2 Encapsulation

Encapsulation is about delimiting scope. We have looked at two manifestations: One is the ‘horizontal’ encapsulation of layers, i.e., the scope of influence of each layer, and the other is the ‘vertical’ encapsulation of layers, i.e., which entities and functions inside a layer is

exposed to the users of the layer ('above' the layer).

Horizontal Encapsulation Traditionally, in the classic Internet, there was one global layer and each endpoint was reachable from all other endpoints. Additionally, separate 'private' TCP/IP networks could exist decoupled from this 'public' Internet. If an endpoint belonged to the public Internet, it was itself publicly available; if an endpoint belonged to a private TCP/IP network, it could only access that private network and not the public Internet. Thus, there was a clear encapsulation of each network: One global, public Internet, and several local, private networks. While each network ran in L3, they were different instances of L3 layers.

In the modern Internet, it is more complicated. While there is still, in principle, only one global public Internet, NAT means that private networks can be attached to it so that the endpoints in the private networks can access the endpoints in public Internet. An endpoint outside a private network cannot access an endpoint inside the private network, except if the private network allows it, which happens, if a reply to a connection attempt from the inside is expected from the outside. Firewalls, VPN, tunnels, and SDN further complicates the picture, and the scope of L3 from the point of an endpoint depends on where in the network the endpoint is, possibly its address, the configuration of the part of the network it belongs to and that part's interaction with other parts of the network, and many other factors. Thus, the horizontal encapsulation for L3 is not well defined, varies depending on network location, configuration of routers, generally changes over time and might depend on network state and many outside factors.

In RINA, layers are defined as DIFs, and each have a clearly delimited horizontal scope. Only members of a DIF may send and receive PDUs on the DIF. There are clear processes governing how to join and leave a DIF, and only authorised entities may do so. Concepts from TCP/IP, such as NAT, complicated routing configurations, tunnels, firewalls, etc., that are applied in order to adjust horizontal scope is not relevant in RINA.

Like TCP/IP, NDN has, in principle, one layer with a global scope. Also like TCP/IP, scope is limited by adjusting routing state to prevent certain PDU from being forwarded under certain circumstances. The result is that NDN layers, like TCP/IP, do not have a clear and well-defined scope. Instead, the scope depends on the interplay between names, routing state, and network configuration.

Chapter 8

Conclusions

In this dissertation, we assess several IAs with a focus on five central properties: flexibility, mobility, scalability, security and simplicity, and their relation to four principles of systems design: extensionality, generalisation, encapsulation and indirection. Through analysis of the design of three internet architectures (IAs), we find issues with several of the properties, and that the issues can be traced back to the neglect of one or more of the principles. Conversely, we also find that the IA that have achieved desirable properties have done so through the deliberate application of these principles. Our findings are summarised in Table 8.1.

IA	Principles				Properties				
	Ext	Gen	Enc	Ind	Flx	Mob	Scl	Sec	Sim
TCP/IP	-1	-1	0	-1	-1	0	0	-2	-1
Modern	-1	-1	0	0	-1	1	-1	0	-2
NDN	-2	-2	-2	-2	-2	0	-2	1	0
RINA	2	2	2	2	2	2	2	2	1

Table 8.1: Principles and properties scores.

These findings suggest that principles of systems design apply as well to networks and internets as they do to any large system. This is not surprising: We can not think of a good reason why they should not. Our findings suggest that neglecting the principles leads to serious and fundamental issues that cannot be remedied after the system has become operational because other systems, as well as its own subcomponents, will then depend on its flawed design.

We argue that an IA, to be successful in the long run,¹ must pay attention to its application of core systems design principles.

¹While TCP/IP has been *the* Internet architecture for decades, and measured by popularity successful until now, we are currently stuck with its fundamental limitations, which limits its usage for the future. As such, TCP/IP is not successful in the long run.

8.1 Contributions

This dissertation provide the following contributions:

1. A case for more discipline and adherence to core principles in network architecture design.

based on:

2. Comparisons and assessments of the following IAs focusing on selected properties and relations to systems design principles:
 - The Classical Internet, including TCP/IP, DNS and the POSIX socket API.
 - Modern Internet Augmentations, including SDN, NFV, loc/id, and MPLS.
 - The following FIAs: NDN and RINA.
3. A taxonomy of the relation between selected properties of IAs and core systems design principles.
4. A brief summary of the state-of-the-art of existing IA comparisons.

We believe this work can influence further critical analyses of current and future IAs based on fundamental and universal software architecture and design principles.

8.2 Perspective

The failure to apply universal principles of systems design to the Internet and FIAs despite their irrevocable status as ‘systems’ and the fundamental and serious issues that result from this, points to a perhaps deeper issue: Somehow an assumption has arisen that the fundamentals of systems design do not apply to the field of network architecture. Based on the results presented in this dissertation, we argue this is wrong: The fundamental principles of our taxonomy applies to internet design as well as to any other software system.

Replacing TCP/IP with RINA would improve the properties of the Internet on all parameters defined in this dissertation. However, being better is not enough—if an existing solution is ‘good enough’ compared to a new solution, the gains of a new, better solution may not measure up to the pain of changing to the new solution.

With regards to a change from the current TCP/IP IA to RINA, we believe that change will only occur when a) the economic benefits of changing to RINA exceeds the benefits of staying on TCP/IP , and b) internet service providers (ISPs) are well-informed enough about the differences, the pros and the cons of each IA to make such a choice . Our hope is that this work may help with Item b), while RINA features and ability to either run on top of TCP/IP as an overlay, or allow TCP/IP to run on top of it will allow a gradual and economic viable

path for Item a).

References

- Afanasyev, Alexander, Priya Mahadevan, Ilya Moiseenko, Ersin Uzun, and Lixia Zhang (2013). "Interest Flooding Attack and Countermeasures in Named Data Networking". In: *IFIP Networking*, pp. 26–31 (cit. on pp. 110, 164).
- Afanasyev, Alexander, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang (2015). "SNAMP: Secure Namespace Mapping to Scale NDN Forwarding". In: *2015 IEEE Conference on Computer Communications Workshops*, pp. 281–286 (cit. on pp. 109, 111, 113, 114, 162, 163, 166).
- Anand, Ashok, Fahad Dogar, et al. (2011). "XIA: An Architecture for an Evolvable and Trustworthy Internet". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks - HotNets-X* (cit. on pp. 27, 33).
- Atkinson, RJ and SN Bhatti (2012). *Identifier-Locator Network Protocol (ILNP) Architectural Description*. RFC 6740 (Experimental). Internet Engineering Task Force (cit. on p. 14).
- Baran, Paul (1964). "On Distributed Communications Networks". In: *IEEE Transactions on Communications Systems* 12.1, pp. 1–9 (cit. on p. 11).
- Black, Andrew, Norman Hutchinson, Eric Jul, and Henry Levy (1986). "Object Structure in the Emerald System". In: *SIGPLAN Notices* 21.11, pp. 78–86 (cit. on p. 170).
- Boguñá, M, Fragkiskos Papadopoulos, and Dmitri Krioukov (2010). "Sustaining the internet with hyperbolic mapping". In: *Nature Communications* 9. arXiv: [arXiv:1009.0267v2](https://arxiv.org/abs/1009.0267v2) (cit. on p. 163).
- Bolt Beranek and Newman, Inc (1981). *A history of the ARPANET: The first decade*. Tech. rep. 4799. Arlington, Virginia: DARPA (cit. on pp. 10, 11).
- Borman, D., B. Braden, V. Jacobson, and R. Scheffenegger (2014). *TCP Extensions for High Performance*. RFC 7323 (Proposed Standard). Internet Engineering Task Force (cit. on p. 57).
- Brooks, Frederick P (1987). "Essence and Accidents of Software Engineering". In: *IEEE Computer* 20.4, pp. 10–19 (cit. on p. 17).
- Bush, R. and D. Meyer (2002). *Some Internet Architectural Guidelines and Philosophy*. RFC 3439 (Informational). Internet Engineering Task Force (cit. on pp. 32, 34).
- Carpenter, B. (1996). *Architectural Principles of the Internet*. RFC 1958 (Informational). Updated by RFC 3439. Internet Engineering Task Force (cit. on pp. 31, 32).

- CCNx (2016). *CCNx specifications*. <http://www.ccnx.org/specifications/> (cit. on pp. 106, 112, 151, 155).
- Cerf, V., Y. Dalal, and C. Sunshine (1974). *Specification of Internet Transmission Control Program*. RFC 675. Internet Engineering Task Force (cit. on pp. 50, 52).
- Cerf, Vinton G, I Introduction, Edward Cain, The Dod, and Internet Architecture (1983). "The DoD Internet Architecture Model". In: *Computer Networks* 7, pp. 307–318 (cit. on pp. 10, 12).
- Chiosi, Margaret, Don Clarke, et al. (2012). *Network Functions Virtualisation: An Introduction Benefits Enablers Challenges & Call for Action*. Tech. rep. 1. Darmstadt, Germany: European Telecommunications Standards Institute (cit. on p. 14).
- Clark, David Dana (1988). "The design philosophy of the DARPA Internet protocols". In: *ACM SIGCOMM Computer Communication Review* 18.4, pp. 106–114 (cit. on pp. 10, 12, 32, 65).
- Croft, W.J. and J. Gilmore (1985). *Bootstrap Protocol*. RFC 951 (Draft Standard). Updated by RFCs 1395, 1497, 1532, 1542, 5494. Internet Engineering Task Force (cit. on p. 42).
- Dannewitz, Christian, Dirk Kutscher, Börje Ohlman, Stephen Farrell, Bengt Ahlgren, and Holger Karl (2013). "Network of Information (NetInf) – An Information-Centric Networking Architecture". In: *Computer Communications* 36.7, pp. 721–735 (cit. on pp. 2, 13, 34, 103).
- Davies, D. W., K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson (1967). "A digital communication network for computers giving rapid response at remote terminals". In: *Proceedings of the ACM symposium on Operating System Principles - SOSOP '67*. Gatlingburg, Tennessee, pp. 2.1–2.17 (cit. on p. 11).
- Day, John (2008a). *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall (cit. on pp. iii, 8–12, 15, 17, 20, 21, 32, 59, 61, 65–67, 72, 75, 76, 83, 86, 89, 95).
- (2008b). "Why Loc/Id Split Isn't the Answer". In: (cit. on pp. 109, 113, 163, 176).
- (2011). "Back to the future: a journey from science to craft ... and back?" In: *Keynote from the Future Internet Assembly 2011: Achievements and Technological Promises*. May. Budapest (cit. on p. 16).
- (2016). "How Naming and Addressing (and Routing) Are Supposed to Work" (cit. on pp. 26, 51).
- Day, John, Steve Bunch, Eleni Trouva, Eduard Grasa, and Peter L. DeWolf (2013). *The RINA specification handbook v0.1*. Tech. rep. January. Pouzin Society (cit. on pp. 9, 65, 66, 86, 89).
- Day, John, Ibrahim Matta, and Karim Mattar (2008a). "Networking is IPC". In: *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*. New York, New York, USA: ACM Press, pp. 1–6 (cit. on pp. 8, 65, 117).
- (2008b). "'Networking is IPC': A Guiding Principle to a Better Internet". In: *ACM CoNEXT '08* (cit. on pp. 65, 117).

- Day, John, Eleni Trouva, et al. (2011). "Bounding the router table size in an ISP network using RINA". In: *2011 International Conference on the Network of the Future*. IEEE, pp. 57–61 (cit. on pp. 92, 163).
- Day, John and Hubert Zimmermann (1983). "The OSI Reference Model". In: *Proceedings of the IEEE* 71.12, pp. 1334–1340 (cit. on pp. 10, 12, 17).
- Deering, S. and R. Hinden (1998). *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. Internet Engineering Task Force (cit. on pp. 33, 60).
- Dijkstra, Edsger W (1968). "The Structure of the THE-Multiprogramming System". In: *Communications of the ACM* 11.5, pp. 341–346 (cit. on pp. 12, 98).
- Droms, R. (1993). *Dynamic Host Configuration Protocol*. RFC 1531 (Proposed Standard). Obsoleted by RFC 1541. Internet Engineering Task Force (cit. on p. 42).
- Farinacci, D., V. Fuller, D. Meyer, and D. Lewis (2013). *The Locator/ID Separation Protocol (LISP)*. RFC 6830 (Experimental). Internet Engineering Task Force (cit. on pp. 14, 33, 109).
- Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine (cit. on pp. 7, 8, 13).
- Fletcher, John G. and Richard W. Watson (1978). "Mechanisms for a reliable timer-based protocol". In: *Computer Networks* (1976) 2.4-5, pp. 271–290 (cit. on pp. 65, 87).
- Floyd, S. (2002). *General Architectural and Policy Considerations*. RFC 3426 (Informational). Internet Engineering Task Force (cit. on p. 99).
- Ford, A., C. Raiciu, M. Handley, and O. Bonaventure (2013). *TCP Extensions for Multipath Operations with Multiple Addresses*. RFC6824. Request for Comments. IETF (cit. on p. 161).
- Forsberg, D., Y. Ohba, B. Patil, H. Tschofenig, and A. Yegin (2008). *Protocol for Carrying Authentication for Network Access (PANA)*. RFC 5191 (Proposed Standard). Updated by RFC 5872. Internet Engineering Task Force (cit. on p. 42).
- Ghods, Ali, Teemu Koponen, Barath Raghavan, Scott Shenker, Ankit Singla, and James Wilcox (2011a). "Information-centric networking: Seeing the Forest for the Trees". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks - HotNets-X*. New York, New York, USA: ACM Press, pp. 1–6 (cit. on p. 16).
- (2011b). "Intelligent Design Enables Architectural Evolution". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks - HotNets-X* (cit. on p. 15).
- Gilligan, R., S. Thomson, J. Bound, and W. Stevens (1999). *Basic Socket Interface Extensions for IPv6*. RFC 2553 (Informational). Obsoleted by RFC 3493, updated by RFC 3152. Internet Engineering Task Force (cit. on pp. 43, 52, 169).
- Goldstein, Fred, John Day, and John Nolan (2011). "The Last Waltz and Moving Beyond TCP/IP". In: *The Journal of the Institute of Telecommunications Professionals* 3, pp. 42–50 (cit. on p. 117).
- Gont, F. and S. Bellovin (2012). *Defending against Sequence Number Attacks*. RFC 6528 (Proposed Standard). Internet Engineering Task Force (cit. on p. 57).

- Haleplidis, E., K. Pentikousis, S. Denazis, J. Hadi Salim, D. Meyer, and O. Koufopavlou (2015). *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426 (Informational). Internet Engineering Task Force (cit. on pp. 14, 31).
- Hoffman, P. and S. Harris (2006). *The Tao of IETF - A Novice's Guide to the Internet Engineering Task Force*. RFC 4677 (Informational). Obsoleted by RFC 6722. Internet Engineering Task Force (cit. on pp. 4, 48).
- Hoque, A K M Mahmudul, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang (2013). "NLSR: Named-data Link State Routing Protocol". In: *ACM SIGCOMM workshop on Information-centric networking*, pp. 15–20 (cit. on pp. 15, 104).
- IEEE 802.11 (2007). *IEEE 802.11-2007: IEEE Standard for Information technology – Telecommunications and information exchange between systems Local and metropolitan area networks – Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. <http://standards.ieee.org/getieee802/download/802.11-2012.pdf> (cit. on p. 42).
- IEEE 802.1X (2010). *IEEE 802.1X-2010: IEEE Standard for Local and metropolitan area networks – Port-Based Network Access Control*. <http://standards.ieee.org/getieee802/download/802.1X-2010.pdf> (cit. on p. 42).
- International Standardization Organization (ISO) (2002). "Information technology - Telecommunications and information exchange between systems - Intermediate System to Intermediate System intra-domain routing information exchange protocol for use in conjunction with the protocol for providing the connectionless". In: *ISO/IEC 10589*, p. 201 (cit. on p. 14).
- ITU-T (1994). "ISO/IEC 7498-1:1994(E), Data Networks and open system communications – Open Systems Interconnection – Model and Notation". In: *Series X: Data Networks and Open System Communications X.200.07/94* (cit. on pp. 9, 65).
- Jacobson, Van, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard (2009). "Networking Named Content". In: *ACM SIGCOMM CoNEXT*, pp. 1–12 (cit. on pp. 2, 13, 15, 22, 32, 103–105, 113, 149, 150, 155, 173).
- Jain, Raj, K. K. Ramakrishnan, and Dah-Ming Chiu (1988). *Congestion avoidance in computer networks with a connectionless network layer*. Tech. rep. DEC-TR-506: Digital Equipment Corporation (cit. on p. 65).
- Jul, Eric, Henry Levy, Norman Hutchinson, and Andrew Black (1988). "Fine-grained Mobility in the Emerald System". In: *ACM Trans. Comput. Syst.* 6.1, pp. 109–133 (cit. on p. 170).
- Kaufman, C., P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen (2014). *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296 (INTERNET STANDARD). Updated by RFCs 7427, 7670. Internet Engineering Task Force (cit. on p. 47).
- Kent, S. and K. Seo (2005). *Security Architecture for the Internet Protocol*. RFC 4301 (Proposed Standard). Updated by RFCs 6040, 7619. Internet Engineering Task Force (cit. on p. 47).
- Koponen, Teemu, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica (2007). "A Data-Oriented (and Beyond) Network

- Architecture". In: *ACM SIGCOMM Computer Communication Review* 37.4, pp. 181–192 (cit. on pp. 2, 13, 15, 34, 103).
- Koponen, Teemu, Scott Shenker, et al. (2011). "Architecting for innovation". In: *ACM SIGCOMM Computer Communication Review* 41.3, pp. 24–36 (cit. on p. 15).
- Kurose, James F. and Keith W. Ross (2013). *Computer Networking*. Sixth Int. Pearson Education (cit. on pp. 10, 37, 56, 59, 60, 117).
- Larzon, L-A., M. Degermark, S. Pink, L-E. Jonsson, and G. Fairhurst (2004). *The Lightweight User Datagram Protocol (UDP-Lite)*. RFC 3828 (Proposed Standard). Updated by RFC 6335. Internet Engineering Task Force (cit. on p. 51).
- Leffler, Samuel J., Robert S. Fabry, and William N. Joy (1983). *A 4.2Bsd Interprocess Communication Primer*. Tech. rep. Berkeley, CA, USA (cit. on p. 50).
- Meyer, David and Darrel Lewis (2009). *Architectural Implications of Locator/ID Separation*. draft-meyer-loc-id-implications-01. Internet Engineering Task Force (cit. on pp. 109, 113, 163, 176).
- Mosko, Marc (2014). *CCNx 1.0 Bidirectional Streams*. Tech. rep. Palo Alto, CA, United States: Palo Alto Research Center, pp. 1–4 (cit. on pp. 158, 166).
- Moskowitz, R., T. Heer, P. Jokela, and T. Henderson (2015). *Host Identity Protocol Version 2 (HIPv2)*. RFC 7401 (Proposed Standard). Internet Engineering Task Force (cit. on p. 14).
- Moskowitz, R. and P. Nikander (2006). *Host Identity Protocol (HIP) Architecture*. RFC 4423 (Informational). Internet Engineering Task Force (cit. on p. 14).
- Moy, J. (1998). *OSPF Version 2*. RFC 2328 (INTERNET STANDARD). Updated by RFCs 5709, 6549, 6845, 6860, 7474. Internet Engineering Task Force (cit. on p. 14).
- NDN-CCL (2016). *NDN Common Client Libraries*. <http://named-data.net/codebase/platform/ndn-cc1/> (cit. on pp. 106, 112, 151).
- Newton, Isaac (2014). *The Principia: Mathematical Principles of Natural Philosophy*. University of California Press (cit. on p. 32).
- Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12, pp. 1053–1058 (cit. on pp. 2, 18, 29, 30, 33).
- Parnas, D L, P C Clements, and D M Weiss (1985). "The Modular Structure of Complex Systems". In: *IEEE Transactions on Software Engineering* SE-11.3, pp. 259–266 (cit. on pp. 2, 29, 30, 33).
- Perkins, C. (2010). *IP Mobility Support for IPv4, Revised*. RFC 5944 (Proposed Standard). Internet Engineering Task Force (cit. on p. 33).
- Popa, Lucian, Ali Ghodsi, and Ion Stoica (2010). "HTTP as the Narrow Waist of the Future Internet". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (cit. on p. 13).

- POSIX.1-2008, 2013 version (2013). *POSIX.1-2008, 2013 version / The Open Group Base Specifications Issue 7/IEEE Std 1003.1TM, 2013 Edition*. <http://pubs.opengroup.org/onlinepubs/9699919799/> (cit. on p. 40).
- Postel, J. (1980a). *DoD standard Internet Protocol*. RFC 760. Obsoleted by RFC 791, updated by RFC 777. Internet Engineering Task Force (cit. on p. 50).
- (1980b). *DoD standard Transmission Control Protocol*. RFC 761. Obsoleted by RFC 793. Internet Engineering Task Force (cit. on p. 50).
- (1980c). *User Datagram Protocol*. RFC 768 (INTERNET STANDARD). Internet Engineering Task Force (cit. on p. 50).
- (1981a). *Internet Protocol*. RFC 791 (INTERNET STANDARD). Updated by RFCs 1349, 2474, 6864. Internet Engineering Task Force (cit. on p. 50).
- (1981b). *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). Updated by RFCs 1122, 3168, 6093, 6528. Internet Engineering Task Force (cit. on pp. 50, 57).
- Pouzin, Louis (1973). “Presentation and Major Design Aspects of the CYCLADES Computer Network”. In: *Proceedings of the Third ACM Symposium on Data Communications and Data Networks: Analysis and Design*, pp. 80–87 (cit. on pp. 10, 11, 17, 65).
- Raghavan, Barath, Teemu Koponen, Ali Ghodsi, Vjeko Brajkovic, and Scott Shenker (2012). *Making the Internet More Evolvable*. Tech. rep. September. International Computer Science Institute, UC Berkeley (cit. on pp. 15, 62).
- Raghavan, Barath, Teemu Koponen, Ali Ghodsi, Martín Casado, Sylvia Ratnasamy, and Scott Shenker (2012). “Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure”. In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks - HotNets-XI*, pp. 43–48 (cit. on pp. 15, 26, 33).
- Saltzer, J. (1993). *On the Naming and Binding of Network Destinations*. RFC 1498 (Informational). Internet Engineering Task Force (cit. on pp. 26, 65).
- Saltzer, J. H. (1978). “Naming and Binding of Objects”. In: *Lecture Notes in Computer Science*. Ed. by R. Bayer, R. M. Graham, and G. Seegmüller. Vol. 60. Operating Systems, An Advanced Course. Springer Berlin Heidelberg, pp. 99–208 (cit. on p. 65).
- Siganos, Georgos (2006). “Jellyfish: A Conceptual Model for the AS Internet Topology”. In: *Communications and Networks* 8.3, pp. 339–350 (cit. on p. 55).
- Srinivasan, C., A. Viswanathan, and T. Nadeau (2004). *Multiprotocol Label Switching (MPLS) Label Switching Router (LSR) Management Information Base (MIB)*. RFC 3813 (Proposed Standard). Internet Engineering Task Force (cit. on p. 1).
- Stevens, W. P., G. J. Myers, and L. L. Constantine (1974). “Structured design”. In: *IBM Systems Journal* 13.2, pp. 115–139 (cit. on pp. 2, 18, 29, 30, 33).
- Stewart, R. (2007). *Stream Control Transmission Protocol*. RFC 4960 (Proposed Standard). Updated by RFCs 6096, 6335, 7053. Internet Engineering Task Force (cit. on p. 51).
- Szabó, Róbert, Balázs Sonkoly, and Mario Kind (2014). *UNIFY: Deliverable 2.2: Final Architecture*. Tech. rep. EU FP7 UNiFY (cit. on p. 14).

- Tanenbaum, Andrew S. and David J. Wetherall (2011). *Computer Networks*. Fifth Int. Pearson Education (cit. on pp. 10, 33, 37, 59, 60, 69, 78, 106, 107, 112, 117).
- Tennenhouse, D.L., J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden (1997). "A survey of active network research". In: *IEEE Communications Magazine* 35.1, pp. 80–86 (cit. on p. 31).
- Trouva, Eleni, Eduard Grasa, John Day, Ibrahim Matta, Lubomir T Chitkushev, Patrick Phelan, Miguel Ponce De Leon, and Steve Bunch (2011). "Is the Internet an Unfinished Demo? Meet RINA!" In: *TERENA Networking Conference (TNC)*. Trans-European Research and Education Networking Association, pp. 1–12 (cit. on p. 65).
- Vrijders, Sander, Dimitri Staessens, Didier Colle, Francesco Salvestrini, Eduard Grasa, Miquel Tarzan, and Leonardo Bergesio (2014). "Prototyping the recursive internet architecture: the IRATI project approach". In: *IEEE Network* 28.2, pp. 20–25 (cit. on p. 118).
- Wang, Lan, A K M Mahmudul Hoque, Cheng Yi, Adam Alyyan, and Beichuan Zhangy (2012). *OSPFN: An OSPF Based Routing Protocol for Named Data Networking*. Tech. rep. (cit. on pp. 15, 104).
- Watson, Richard W. (1981). "Timer-based mechanisms in reliable transport protocol connection management". In: *Computer Networks* (1976) 5.1, pp. 47–56 (cit. on pp. 65, 87).
- Wittenstein, L. (2012). *Tractatus Logico Philosophicus*. Unabridged Start Publishing LLC. Start Publishing LLC (cit. on p. 32).
- Xylomenos, George, Christopher N. Ververidis, Vasilios a. Siris, Nikos Fotiou, Christos Tsilopoulos, Xenofon Vasilakos, Konstantinos V. Katsaros, and George C. Polyzos (2014). "A Survey of Information-Centric Networking Research". In: *IEEE Communications Surveys & Tutorials* 16.2, pp. 1024–1049 (cit. on p. 15).
- Zhang, Lixia, Alexander Afanasyev, et al. (2014). "Named data networking". In: *ACM SIGCOMM Computer Communication Review* 44.3, pp. 66–73 (cit. on pp. 2, 15, 32, 34, 103, 104, 106, 108, 111, 113, 144, 149, 150, 157, 158, 173).
- Zhang, Lixia, Deborah Estrin, et al. (2010). *Named Data Networking (NDN) Project*. Tech. rep. (cit. on pp. 103–105, 150, 154, 157, 158).

Glossary

CYCLADES The CYCLADES network . 16

Acronyms

AE application entity. 10, 50, 66, 72, 73, 80, 81, 89, 91, 97, 124, 125, 127, 129–131, 133, 136, 137, 140, 202

AN application name. 97, 100, 101, 129, 170

AP application process. 10, 44, 51, 53, 55, 66–72, 75, 79, 80, 88, 93, 105, 124, 125, 129, 130, 139, 140, 150, 164

API application protocol interface. iii, 14–16, 22, 23, 37, 40, 41, 43–47, 50–54, 58–63, 65, 66, 72, 78, 89, 91, 94, 95, 97, 99–101, 106, 107, 112–114, 117–119, 121, 123, 124, 136, 139, 141–144, 151–153, 155–157, 159, 160, 166, 169, 171–175, 180, 203

ARP Address Resolution Protocol. 40, 58, 128, 174

ARPANET the U.S. Department of Defence's Advanced Research Projects Agency Network. 10–12, 201

AS autonomous system. 32, 37

ASN.1 Abstract Syntax Notation One. 88

BGP Border Gateway Protocol. 1

BOOTP Bootstrap Protocol. 42

BSD Berkeley Software Distribution. 40

CCN content centric networking. 2, 13, 15, 17, 32, 103, 112, 144, 155

CCNx . 106, 107, 112, 113, 151, 155, 160, 173, 174

CDAP the common distributed application protocol. 75, 83, 88, 95, 99, 166

CDN content delivery network. 165, 166

- CEID** connection endpoint identifier. 43, 100, 101, 124, 129, 131, 133, 134, 136–138, 171, 173, 176
- CID** connection identifier. 73, 80, 81, 84–87, 97, 169, 170
- CRC** cyclic redundancy check. 84, 85, 88
- DAF** distributed application facility. 156–158
- DAG** directed acyclic graph. 15, 25
- DCA** the U.S. Defense Communications Agency. 11
- DDoS** distributed DoS. *see* DoS
- DHCP** the Dynamic Host Configuration Protocol. 42
- DHT** distributed hash table. 3
- DIF** distributed IPC facility. 69–72, 74, 78–81, 84, 87–89, 91, 93, 96, 97, 100, 119, 121, 131, 133, 136–139, 156–158, 177, 202
- DIP** (distributed) IPC process. 70–76, 78–81, 83, 84, 87–89, 93, 94, 96, 97, 130, 133, 136–138, 140, 143, 156, 161, 164, 172, 202
- DNS** Domain Name System. iii, 1, 37, 38, 42–46, 52, 53, 57, 61, 62, 95, 108, 113, 118, 121, 125, 141, 144, 154, 169, 171, 175, 176, 180
- DNSSEC** secure DNS. *see* DNS
- DONA** the Data-Oriented Network Architecture. 2, 3, 13, 15, 34, 103
- DoS** denial-of-service. 47, 93, 164
- DTCP** the Data Transfer Control Protocol. 85, 86, 167, 203
- DTN** delay or disruption tolerant networking. 13
- DTP** the Data Transfer Protocol. 85, 86, 167, 203
- EFCP** the Error and Flow Control Protocol. 71–76, 78–81, 83–87, 89, 95, 99, 129, 131, 133, 136–139, 142, 143, 159, 167, 168, 202
- EID** endpoint identifier. 41, 44–48, 51–55, 57, 59–62, 73, 80, 81, 94, 97, 124, 125, 127, 136, 139–142, 159–162, 201

Ethernet IEEE 802.3. 8, 14, 38–40, 57, 58, 95, 107, 113, 125, 127–129, 137, 143, 152, 167, 168, 174

ETR egress TR. *see* TR

FA flow allocator. 80

FAI flow allocator instance. 80, 81, 97, 119, 129–131, 133, 137, 140, 144, 170

FEID flow endpoint identifier. 42, 43, 100, 124, 129, 131, 169, 170, 176, 201

FIA future IA. iii, 2, 13, 15–17, 34, 117, 180, *see* IA

FIB forwarding information base. 104, 105, 140, 143, 144, 170, 176

FID flow identifier. 169, 170

FQDN fully qualified domain name. 43, 57, 62, 176

FSM finite state machine. 20–24

FTP the File Transfer Protocol. 38, 99

GPB Google Protocol Buffers. 88

GRE Generic Routing Encapsulation. 174

HIP the Host Identifier Protocol. 1, 14, 113

HTTP HyperText Transfer Protocol. 13, 38, 88, 95

HTTPS Secure HTTP. *see* HTTP

IA internet architecture. iii, iv, vii, 1–5, 7–10, 13–17, 26–31, 33–35, 102, 115, 161–168, 172, 174, 179, 180, 205

IAP the IPC access protocol. 70–72, 74, 76, 78, 202

ICN information centric networking. 2, 13–17, 34, 103, 151

ICSI the International Computer Science Institute at UC Berkeley. v

IETF Internet Engineering Task Force. iii, 4, 48

IKE Internet Key Exchange. 47

ILNP the Identifier-Locator Network Protocol. 1, 14, 60, 113

IMAP the Internet Message Access Protocol. 38

IMP Interface Message Processor. 11, 201

IP Internet Protocol. 12, 14, 15, 27, 39, 40, 42–48, 50–58, 60–62, 92, 95, 100, 104, 105, 107, 112, 113, 124, 125, 127–129, 134–137, 139, 141–144, 149, 150, 152, 154, 160, 163, 166, 171, 174, 176, 204

IPC inter-process communication. 8, 16, 40, 52, 65–73, 75, 76, 80, 88, 89, 91, 94, 95, 98, 99, 113, 142, 149–151, 156–158, 165, 166, 173, 175, 202

IPsec IP Security. 47, 60, 164

IPv4 IP version 4. iii, 37, 38, 52, 58, 60, 141, 167, 168, 170, 174, *see* IP

IPv6 IP version 6. iii, 1, 33, 38, 52, 58, 60–62, 141, 162, 163, 170, 174, *see* IP

IS-IS Intermediate System to Intermediate System. 14, 61

ISP internet service provider. 180

ITR ingress TR. *see* TR

JSON JavaScript Object Notation. 88

L1 layer 1. 38

L2 layer 2. 38, 39, 42, 45, 57, 58, 98, 107, 137, 138, 142, 167

L3 layer 3. 38, 39, 42, 45, 47, 57, 58, 60, 98, 106, 107, 134, 136–138, 142, 167, 171, 177

L4 layer 4. 38, 39, 45, 57, 58, 107, 134, 136–138, 142, 167, 171

L5 layer 5. 38, 136

LAN local area network. 38–40, 91, 98

LEID LISP endpoint identifier. *see* LISP

LISP the Locator/Identifier Separation Protocol. iii, 1, 14, 33, 60, 109, 113

loc/id the location/identifier split. 1, 14, 60, 109, 111, 113, 162, 163, 166, 169, 176, 180

MAC Media Access Control. 39, 40, 174

MF MobilityFirst. 2, 3

MIP Mobile IP. iii, 1, 33, 60, 109, 162, 163, 176

MPLS Multi-Protocol Label Switching. 1, 37, 52, 58, 95, 142, 143, 174, 180

MTU maximum transmission unit. 151

mux multiplexing task. 73–75, 78, 80, 202

NAT network address translation. 37, 57, 60, 134, 171, 177

NCP the Network Control Program. 11

NDN Named Data Networking. iii, 2, 15, 17, 31, 32, 34, 103–115, 144, 149–168, 171, 173–177, 179, 180, 203–205

NDN-CCL . 106, 107, 160, 173, 174

NetInf the Network of Information. 2, 3, 13, 34

NFV Network Function Virtualization. 1, 14, 15, 180

NI Named Information. 103

NIC network interface card. 39, 60, 149, 150

NRS name resolution system. 46, 52, 55, 57, 91, 111, 141, 159, 160, 170, 172, 176

OS operating system. 3, 12, 13, 26, 40, 41, 52–54, 59, 61, 66, 67, 70, 71, 92, 97, 98, 100, 125, 129, 130, 150, 161, 165

OSI Open Systems Interconnection. 9, 10, 12, 16, 201

OSPF Open Shortest Path First. 14, 61, 104

PCI protocol control information. 9, 40, 73, 79, 82, 87, 88, 96, 128, 129, 131, 133, 134, 137, 138, 156

PDU protocol data unit. 9, 28, 40, 57, 58, 73, 75–79, 82–87, 91, 94, 96, 101, 104–108, 110–115, 118, 127–129, 131–134, 137, 142, 144–148, 150, 151, 153–158, 160, 162, 164–166, 171–174, 177, 202, 204

PID port identifier. 67, 68, 72, 73, 80, 97, 170

- PIT** pending interest table. 105
- PM** protocol machine. 9, 21–23, 39, 40, 44, 47, 50, 51, 53, 54, 58, 59, 61, 72–76, 78–81, 83–86, 89, 124, 125, 127–131, 133, 134, 136–140, 172, 173, 201, 202
- PoA** point of attachment. 14, 26, 27, 39, 42–46, 48, 53–55, 61, 62, 93, 140, 143–150, 154, 160–162, 170, 201, 204
- POSIX** Portable Operating System Interface. 40, 43, 45, 52, 142, 169, 171, 180
- PPT** PDU processing time. 85
- PS** processing system. 9, 10, 53, 54, 66, 67, 70, 73, 75, 140, 149, 202
- QoS** quality of service. 57–59, 67, 73, 78, 84, 85, 87, 91, 94, 99, 101, 107, 139, 142, 143, 152, 159, 165–167, 174, 175
- REST** Representational State Transfer. 13, 88, 95
- RIB** resource information base. 78, 79, 88, 108, 133, 136, 137, 143, 144, 170
- RIEP** resource information exchange protocol. 75, 78, 83
- RINA** the Recursive Internet Architecture. iii–v, 3, 16, 17, 32, 50, 59, 60, 65, 66, 81, 83, 89–102, 108, 117–121, 123, 124, 129–132, 134, 136–144, 151, 152, 156, 158–168, 170, 172, 173, 175–177, 179, 180, 202–205
- RMT** relay and multiplexing task. 78–80, 84, 130, 131, 133, 136–138, 202
- RTP** the Real-time Transport Protocol. 95
- RTT** round-trip time. 85
- SCTP** the Stream Control Transmission Protocol. 51, 99, 159
- SDIA** the Software-Defined Internet Architecture. v, 2, 3, 15, 26
- SDN** Software-Defined Networking. iii, 1, 14, 15, 31, 60–62, 161, 163, 166, 176, 177, 180
- SDU** service data unit. 9, 72, 73, 82, 84, 85, 96, 118, 127–129, 131, 133, 134, 156
- SIP** the Session Initiation Protocol. 57
- SN** sequence number. 84–87

SNAMP Secure Namespace Mapping. 109, 111, 113, 114, 162, 163, 166

SSH the Secure Shell Protocol. 99

TCP Transmission Control Protocol. 12, 38–41, 44, 47, 50, 51, 54, 57–59, 83, 95, 99, 107, 112, 118, 119, 121–129, 131, 136, 139, 143, 152, 159, 167, 168, 174, 201, 203, 205

TCP/IP the Transmission Control Protocol/Internet Protocol protocol suite. iii, vii, 1–3, 10, 12–17, 30–33, 37–40, 42, 44–48, 50, 51, 53–55, 57, 59–61, 89, 93, 95, 106–110, 112, 117, 118, 124, 126, 127, 134, 135, 137–144, 158–169, 171, 173–177, 179, 180, 201, 203–205, *see* TCP & IP

the Web the World Wide Web. 13, 99

TLS Transport Layer Security. 95

UDP User Datagram Protocol. 12, 38–40, 44, 47, 50, 51, 54, 57–59, 83, 95, 99, 107, 113, 118, 139, 143, 152, 159, 167, 168, 174, 201

UDP-Lite the Lightweight User Datagram Protocol. 51

URL Uniform Resource Locator. 43, 169, 176

VPN virtual private network. 81, 91, 98, 101, 164, 171, 177

WiFi IEEE 802.11. 38, 42, 57

WPA2 WiFi Protected Access II. 42

XIA the eXpressible Internet Architecture. 2, 15, 26

List of Figures

- 2.1 The basic ARPANET architecture, consisting of hosts, IMPs, and telephone lines. Adapted from Bolt Beranek and Newman, Inc 1981, Figure 1. 11
- 2.2 The layers of the Internet: TCP/IP reference model layers and OSI layers. Adapted from V. G. Cerf et al. 1983, Figure 4. 12
- 2.3 Two systems communicating via a protocol. 22
- 2.4 The Network PM 23

- 3.1 Our view of the main relations between chosen taxonomy principles and the core principles of modularity and abstractions. 30

- 4.1 TCP/IP model. 38
- 4.2 Flow between applications is equivalent to the flow between sockets. The FEIDs of the flow are the socket identifiers. 43
- 4.3 A TCP/IP stack and application. Applications bind statically to the socket/PM, which binds statically to a PoA. If either the PM or PoA becomes invalidated, the application needs to recreate the bindings. 44
- 4.4 Principles applied or not applied to the Internet architecture, and the consequences thereof. 49
- 4.5 Legend for Figures 4.4, 4.6, 4.7 and 4.9 to 4.11. 49
- 4.6 Flexibility property of the Internet 50
- 4.7 Mobility property of the Internet 53
- 4.8 TCP/IP socket states. For the two types of sockets—TCP or UDP—the possible states are: *u*—the unbound state, where the socket is created but has not been assigned an EID yet; *b*—the bound state, where an EID has been assigned to it, but no peer is bound yet; *a*—the associated state, where the socket is bound to its own EID and associated with its apposite’s EID and state (only relevant for TCP/IP); *t*—the terminated state, after the socket is closed. 54

4.9 Scalability property of the Internet 55

4.10 Security property of the Internet 56

4.11 Simplicity property of the Internet 58

5.1 Local communication between applications **A** and **B**. Adapted from Day
2008a, Figure 6–4. 67

5.2 Applications communicating between two systems 70

5.3 Elements of two applications communicating via a DIF spanning two systems.
Adaptation of Day 2008a, Figure 6–5. 72

5.4 Multiple simultaneous connections between two systems requiring multiple
EFCP PM instances and a mux in each DIP 74

5.5 N directly connected systems 74

5.6 An interim picture of a PS in a network of N directly connected systems. We
now have multiple mediums, each requiring their own mux to multiplex the
PDUs from the EFCP PMs bound to each interface or mux. Adaptation of
Day 2008a, Figure 6–11. 75

5.7 Each medium require their own lower-level IPC management task and IAP
PM, and we start to see a pattern of elements from the upper-level DIP
repeated in a per-medium DIP-like component. Adaption of Day 2008a,
Figure 6–12. 76

5.8 N systems indirectly connected. Systems can be categorised into routers,
which relay PDUs between systems that are not directly connected, and hosts,
which produce and consume PDUs. A system might assume both roles. . . . 77

5.9 A simplified interim router design 77

5.10 Systems in a relayed network 79

5.11 Flow and connections between AEs 80

5.12 High-level picture of a DIP 81

5.13 DIFs constitute the layers in RINA. Here four DIFs are shown: One network-
wide N -DIF, spanning three link $(N-1)$ -DIFs. Each $(N-1)$ -DIF abstracts a
physical medium (which we can consider to be a bottom pseudo-layer). . . . 81

5.14 More elaborate layer layout of an internetwork 82

5.15 The RMT handling error-checking of incoming PDUs, relaying, multiplexing
and demultiplexing 84

5.16	The tightly bound mechanisms form the DTP on the left-hand side, while the loosely bound mechanisms form the DTCP on the right-hand side. The two are tied together by a state vector. The figure is based on Day 2008a, Figure 3-5 and Day, Bunch, et al. 2013, EFCP Spec, Figure 1.	86
5.17	Properties and principles in RINA	90
5.18	Flexibility property in RINA	90
5.19	Mobility Property in RINA	91
5.20	Scalability Property in RINA.	92
5.21	Security Property in RINA	93
5.22	Simplicity Property in RINA	94
5.23	Principles in RINA	96
6.1	An NDN Forwarder. Figure courtesy of Jacobson et al. 2009; Zhang, Estrin, et al. 2010.	105
6.2	NDN Principles and Properties	106
6.3	NDN Flexibility Property	107
6.4	NDN Mobility Property	108
6.5	NDN Scalability Property	109
6.6	NDN Security Property	110
6.7	NDN Simplicity Properties	111
7.1	A consumer-producer API example.	118
7.2	TCP producer-consumer example without mobility.	119
7.3	RINA producer-consumer example without mobility.	120
7.4	TCP producer-consumer example where P changes address after C has obtained its data d . P needs to re-register its name with a new address A' , and C needs to re-establish lost connections using the new address before d' can be written back.	122
7.5	RINA producer-consumer example where P changes address after C has obtained its data d . In RINA, the address change is invisible to applications and is handled entirely by RINA. Therefore, the scenario is similar to Figure 7.3 (except for the address-change incident itself).	123
7.6	Creation of stack state in TCP/IP for producer P and consumer C of the example in Figure 7.2.	126

7.7 Transfer of data from *C* to *P* in TCP/IP. The PDU follows the path of the highlighted components. 127

7.8 Creation of stack state in RINA for producer *P* and consumer *C* of the example in Figure 7.3. 130

7.9 Transfer of data from *C* to *P* in RINA. The PDU follows the path of the highlighted components. 132

7.10 Effects of IP address change in TCP/IP including recovery. Components with dashed borders are being torn down, while components with highlighted borders are being created. The process is explained in details in the text and omitted here. 135

7.11 Connection management and endpoint bindings in TCP/IP and RINA 140

7.12 Example graph showing nodes (*A* through *D*), PoAs (1 through 10), and the nodes' routing tables. 145

7.13 Routing example for a PDU travelling from *A* to *D* with destination address 9. 146

7.14 Routing example for a PDU travelling from *A* towards *D* with destination address 9 with a connection-failure mid-transit. Routing table *A* shows its state before breakdown, while *B*, *C*, and *D* shows their state after breakdown, so that they all show their state at the time of forwarding. Removed entries are shown with black, while updated entries are shown with underlined values. The PDU will be dropped at *B* because its destination address points to a PoA that has disappeared from the network and routing tables. 147

7.15 Routing on node addresses. 147

7.16 Routing on node addresses for PDU travelling from *A* to *D* with a connection-failure mid-transit. The PDU will be re-routed and will reach its destination. . 148

7.17 Inter-application communication in a black-box service architecture like RINA 152

7.18 A first approximation to inter-application communication in the NDN architecture. 153

7.19 A better approximation to inter-application communication in the NDN architecture, now including supporting packet delivery service layers. . . . 153

7.20 Bindings and Indirection in TCP/IP. 161

List of Tables

4.1	Layers in the Internet model.	38
4.2	TCP/IP identifiers, their scope, and their users.	39
5.1	Common, and tightly and loosely bound mechanisms and their properties . .	85
5.2	Principles and Properties Scores for RINA	102
6.1	NDN Principles and Properties Scores.	115
7.1	Differences between TCP and RINA for the producer-consumer example for both scenarios.	124
7.2	The meaning of the components making up Figures 7.6 and 7.7.	125
7.3	The meaning of the components making up Figures 7.8 and 7.9. This is similar to Table 7.2, but updated for RINA.	130
7.4	Differences and similarities between TCP/IP and RINA evident in the traces of Section 7.2.1.	138
7.5	Run Time Flexibility	159
7.6	Flexibility Assessment	161
7.7	Mobility Assessment	162
7.8	Scalability Assessment.	163
7.9	Security Assessment	164
7.10	Functions provided or not provided by various IAs.	165
7.11	Simplicity assessment based on functionality.	167
7.12	Protocols and number of message formats required for end-to-end data transfer.	168
7.13	Simplicity Assessment. The combined assessment is a weighted average, rounded to nearest integer, with application simplicity weighing twice as much as IA simplicity.	168
8.1	Principles and properties scores.	179

