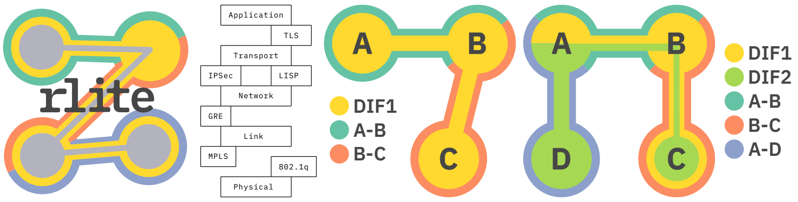
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

rlite: building scalable networks the recursive way

Michal Koutenský*



Abstract
The purpose of this paper is twofold — first, to showcase my contributions to the rlite networking stack, and second, to inform readers about the existence of Recursive InterNetwork Architecture (RINA) and to serve as a quick and easy to read introduction to the architecture and its capabilities. It is well documented that the TCP/IP network architecture suffers from numerous deficiencies and does not meet the demands of modern computing. A great number of these issues are structural and cannot be properly solved by making adjustments to existing protocols or introducing new protocols into the stack. RINA is a clean slate architecture whose aim is to be a more general, robust and dynamic basis for building computer networks. I have extended the rlite implementation with support for policies. With this framework in place, it is possible to have multiple behaviours of components (such as routing), and change between these during runtime. This additional flexibility and simple extensibility greatly benefits both production deployment scenarios as well as research efforts. As policies are crucial for RINA, supporting them is an important milestone for the implementation, and will hopefully foster adoption and accelerate development of RINA as a viable replacement for current Internet.

Keywords: network architecture — RINA — rlite

Supplementary Material: GitHub repository — Pouzin Society

*xkoute04@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The modern computing landscape and its requirements vastly differ from the state of things in early 1970's, where TCP/IP has its roots. Although there has been considerable innovation *on top of* the Internet, *Internet itself* has seen very little change since late 1970's. [1]

It is well documented that the TCP/IP architecture suffers from many problems, which are often structural and cannot be properly solved by making adjustments to the protocol stack. *These problems aren't new*. In fact, some (such as the inability to support proper multihoming) have been known since the ARPANET.[1]

There exist two reasons for why that is the case. As the primary goal was to build a working packet switched network, some known hard problems (like getting naming and addressing right) were given lower priority and left for later. The original ARPANET for example used a simple enumeration of IMP[2] ports for host addresses. At the same time, not much was known about such networks and how they operate. Further research and real world experience was needed to be able to come up with comprehensive solutions.

Therein lies the core problem of the Internet. Unpredictably fast rate of real world adoption and growth

---

[1] The Tinker Air Force Base asked for redundant connection in 1972.

[2] Interface Message Processor, the "router" used in ARPANET.

meant that these issues never got addressed again, as things were working well enough for the time being, while the cost of doing any sort of radical changes to the architecture kept increasing.

The fact that the architecture is insufficient and broken can be plainly seen in the layer model. Traditionally, it is described as a five layer model. However, at this point, five layers only exist in the simplest scenarios and networking textbooks. With introduction of technologies such as 802.1q, MPLS, GRE, IPsec, LISP or TLS, the protocol stack not only increases in size, but has a dynamic number of layers, as new protocols are inserted wherever convenient. This is a clear sign of incomplete architecture and a stopgap approach to solving problems.

Surely, one of the biggest changes of the past thirty years is the introduction of IPv6. However, its adoption leaves a lot to be desired. It has been over 23 years since the original IPv6 RFC [2] was published, and according to statistics from Google [3], around 25% of their customers access their services over IPv6, while five years ago the number was less that 5%.

IPv6 was meant to solve some of the issues of the Internet, chief among them address space exhaustion, but brought with it its own set of issues, the migration process itself being just one of them. It failed to address the problem of naming entities in the network in any significant way, mainly restricting itself to increasing the size of the address space. Many protocols used IPv4 addresses directly, or were designed to only carry IPv4, and thus, they had to be reworked to work with IPv6.

The Loc/Id separation protocol [4] tries to decouple the semantics of identifying a node and locating a node within the network. [5] While this does improve the situation slightly, it only addresses the symptom, not the problem. The naming scheme in TCP/IP is incomplete. IP address does not name a node, it names an interface, just like a MAC address[3] does. Both have a global scope, with MAC space being flat and IP space being hierarchical. In practice, due to address space fragmentation, even this does not hold, and neighbouring interfaces might have radically different addresses. There exist no application names or node addresses[4] within TCP/IP.

Not only have our requirements and use cases changed, but so has our knowledge about computer networks. Decades of real world experience have given us additional insight into the nature of networks and their guiding principles. RINA aims to utilize this knowledge to be a more complete theoretical model that allows network designers to build robust, secure, scalable and manageable networks.

As RINA only defines a theoretical model, it is necessary to implement it to reap practical benefits. rlite is such an implementation. As an open source project, with focus on robustness and clean design, it can already be used to run simple RINA networks, whether directly over Ethernet or WiFi, or as an overlay network over TCP or UDP. It provides an easy way to extend its behaviour with support for runtime policies for all of its components, giving network designers great flexibility in creating solutions that strictly fit their requirements.

My work focuses on policies, which are a critical component of the architecture. The diversity of operating environments and user requirements for computer networks mean that there is no *one-size-fits-all* solution to many problems — it is always a series of trade-offs fitting a particular scenario. [6] This understanding is built into the architecture on an more fundamental level than in TCP/IP. Instead of interchangeable protocols, the logical components themselves have dynamic, configurable parts that can be switched as needed.

I have implemented a framework for registering and switching policies. This allows network administrators to tune the behaviour of the network to fit their requirements. Likewise, it is beneficial for researchers, as it's easy to introduce new behaviour into the network. The policies can be switched and adjusted during runtime, which makes it very convenient to reuse the same network when running experiments, without additional reconfiguration or patching.

The structure of this paper is somewhat unconventional, as the first two sections serve to introduce the reader to RINA in a organic and approachable manner.[5] The first section could be considered transitional, in which we explore the nature of network architectures and layers. Likewise, it contains some criticisms of current networking, to illustrate why a paradigm shift is needed in the first place. The second section builds on the concepts introduced in the first one to present and describe the RINA layer model and how it ties together. The third section addresses rlite and the policy framework therein, followed by a conclusion that summarizes the paper's contents.

---

[3] As it does not help with locating the interface in any way, it might be more precise to call it a name rather than an address.

[4] Host *names* are not topological. It is not possible to e.g. route on them either.

[5] Unfortunately, this paper would make little sense to the reader without this knowledge. Some compromises therefore have to be made, to present everything within this limited space.

2. From a protocol stack to a recursive structure

There are two well known models for how computer networks are structured — the OSI model [7] and TCP/IP[6] [9, 10]. Most undergraduate students of computer science will (*hopefully!*) come into contact with these. However, due to the dominance of TCP/IP in real-world deployments, there is usually not much time spent on explaining OSI and how it differs from TCP/IP.

How do OSI and TCP/IP differ? A common description might look as such:

“OSI is the *seven layer model* and TCP/IP is the *five layer model*, and we use TCP/IP because the Session and Presentation layers were deemed to serve no practical purpose.”

This kind of answer is wholly insufficient for the purposes of this paper, so let’s briefly go through some history. Aside from the details of the models themselves, there are additional questions that should be considered. *Why are the models layered? What is the purpose of the layers? How did the layers in the model arrive to be?*

Andrew Tanenbaum’s textbook on computer networks [8] provides come clues to the differences between the models.

It claims that one of OSI’s biggest contributions is the strict distinction between *services*, *interfaces* and *protocols*. This roughly corresponds to a object-oriented understanding of layers, and allows to easily replace protocols used in the layers. TCP/IP, on the other hand, did not originally make this distinction explicit, which results in some of the problems outlined above.

The OSI model was created before the protocols (and is thus protocol-agnostic), TCP/IP after. In a very real sense, the TCP/IP model is just a description of the protocol stack. The protocols fit the layers perfectly; the issue is that *the model fits only those protocols*. With the addition of a new set of protocols into the (TCP/IP) stack, a new model is required to describe it.

To further complicate the problem, a lot of protocols in TCP/IP depend on certain protocols being in place in the stack under them[7]. Modern solutions and research efforts often breach the layer boundary[8], justified by pragmatism, seeking more information about network state and fine-grained control. Viewing layers as objects, such dependencies on private behaviour instead of communicating through well-defined public interfaces goes against all good software engineering practices.

We have established that a layer is an object providing services to the layers above. *What functions are necessary for network communication? How did their division into layers come to be?*

A lot of inspiration was taken from operating systems and the body of research work done in this field.

> Network communication is interprocess communication, and nothing but IPC.

The end goal of all network communication is for two application processes to communicate. There exists a common set of operations required for IPC to happen — locating the other process, allocating communication resources, etc. The crucial difference is whether these happen within one processing system or are distributed. For networking communication, there needs to exist a distributed facility providing these services to the participating processes. In an ideal scenario, the end process does not know whether it is communicating within one processing system or over a network, and does not care about this fact either.

Both OSI an TCP/IP attempt to distribute the required functions between layers and create a hierarchy. As we can see, OSI did not get it right; the Session and Presentation layers are infamously never used in practice. However, on closer inspection, TCP/IP did not get it right either — the protocol creep in the stack is a de facto layer creep. (See figure 1.)

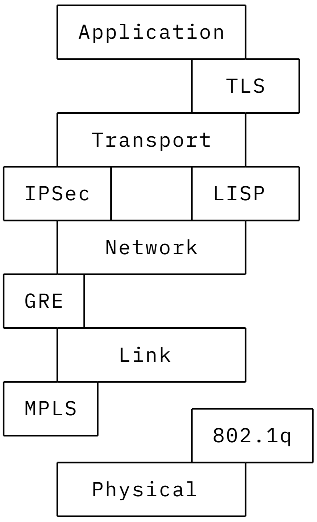


Figure 1. The TCP/IP model with other commonly used protocols. These new protocols do not fit into any particular layer, they exist on layer boundaries, being de facto new layers themselves.

Furthermore, a lot of functions are repeated in the layers, or follow similar patterns. The data link layer is split into Media Access Control (MAC) layer and

---

[6] Often, a hybrid OSI-TCP/IP model is used, as seen in [8]

[7] This fact is even reflected in the name after all.

[8] Think an application accessing TCP window size directly and making decisions based on that.

Logical Link Control (LLC) layer. MAC provides multiplexing and flow control for the physical medium. LLC provides multiplexing and flow control for the logical link. Are not multiplexing and flow control responsibilities of the transport layer? Does UDP provide flow control?

It appears that division into layers *based on function* was not the right model. All layers provide all functions, although a particular layer in a particular network might not require all of them. What makes one layer distinct from another is the *scope of their shared state*.

There exists only one layer, and it recurses.

3. Untangling the recursive architecture

We have arrived at some interesting insights in the previous section. The question now becomes: *can a comprehensive architecture be created using this approach*?

The Recursive InterNetwork Architecture (RINA), as well as the insights from previous section (and much more) were described in John Day's *Patterns In Network Architecture: A Return to Fundamentals*. [1]

In many ways, it signifies a radical departure from networking as understood now. The documents describing a reference model number almost a hundred pages altogether. [11] It is impossible to fully cover it within the limited space provided. This chapter will merely try to familiarize the reader with the main concepts and overall structure.

As stated previously, there is only one layer, and it recurses. First, let's introduce *Distributed Application Facility*. It consists of two or more *Application Processes* which *exchange information* and *maintain shared state*. [12]

A layer is a specialized version of the DAF, a *Distributed IPC Facility*. A DIF is a DAF where the APs do IPC. It provides IPC services to APs of other DIFs via an API. An AP that provides IPC is called an *IPC Process*.

The lowest DIF is the link. Two neighbouring IPCPs, A and B, which are directly connected, can maintain shared state and exchange information.

An (N+1) level common DIF named DIF1 can be created over this (A-B) link DIF. A neighbour C, who shares a common link with B, can join this DIF1, by utilizing the (B-C) link DIF IPC services of B, such as having B relay necessary messages to A over its (A-B) DIF.

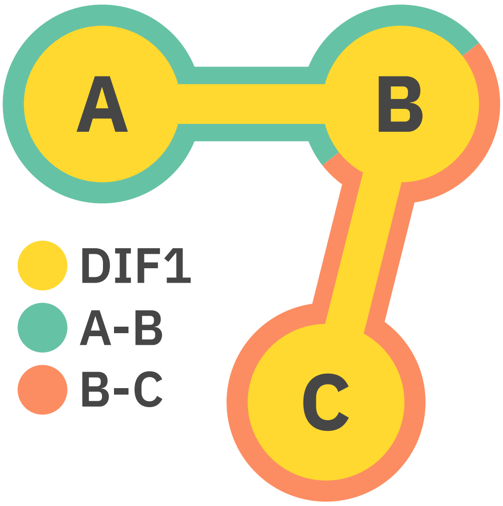


Figure 2. View of the DIFs in the network. An overlay DIF exists over the link connections, allowing all three nodes to communicate with each other.

This can scale indefinitely.[9] If A is a member of another level DIF, C can join this DIF as well, since it can now utilize A's services over the common DIF.

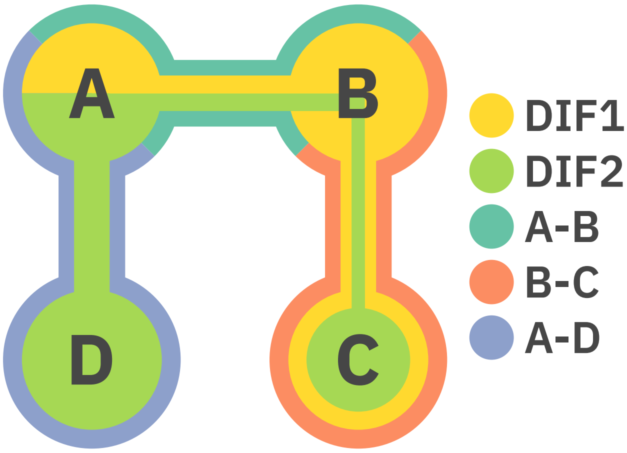


Figure 3. Two non-link DIFs. A and D use the link DIF to join DIF2, while C joins by using its connection with A over DIF1, as it is not a direct neighbour with any member.

As each DIF has limited scope, so does its shared state. The practical implication of this is that *addresses are unique and meaningful only within a DIF*. There is no global address space in vein of IP, as it is not needed. The resembling network is a true *inter*net, not a *cate*net[10]. The size of the address space can be bound, and each DIF can use differently structured addresses that properly reflect the *topological* information.

The mappings between (N) and (N-1) level DIFs are analogous to logical and physical address spaces. Routing is done within a layer (and for each layer), and based on the next hop a suitable (N-1) DIF is picked. Thus, an IPCP can change the (N-1) *Point of Attach-*

---

[9] There are physical constraints, but the architecture itself doesn't impose any.

[10] The Internet is not an internet.

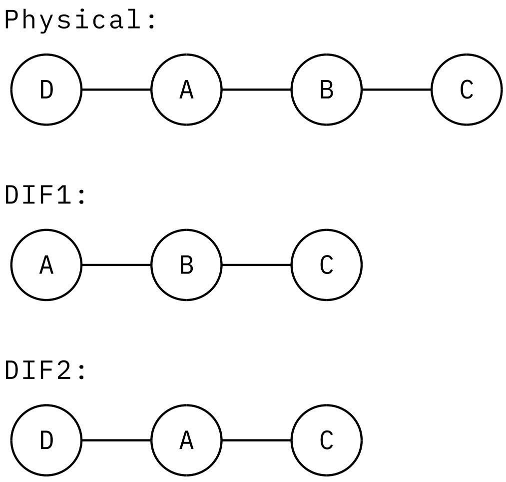


Figure 4. Network graphs showing a) the physical connections between nodes b) the logical connections as known to DIF1 and c) DIF2 respectively.

ment without disturbing its (N) level communication. As such, mobility and multihoming[11] are naturally supported within the architecture.

Each AP has a global *Application Process Name*, allowing uniquely referring to APs by their name, without using node addresses. This dynamic mapping allows applications to have all the benefits described above regarding mobility.

The IPCP consists of various building blocks providing different services. Each of these consists of *mechanisms* and *policies*. Mechanisms are the static, built-in components, whereas policies are variable and change from use-case to use-case. An example of a mechanism might be sending of packet acknowledgements, a policy describes *how and when are ACKs sent*. This is an extremely powerful design decision that gives network designers great freedom and extensibility to design the network in a way that is suitable for the operating environment and the requirements imposed by both applications and e.g. company politics.

To illustrate with an example, the *Flow Allocator* component is responsible for handling requests for allocation of flows, as the name suggests. In TCP/IP, such requests are always accepted. By utilizing policies, the network designer can support and enforce a wide variety of use-cases. A policy can be made to reject the request if there does not exists a path from source to target within the network which has a capacity of at least X Mbps. Another policy could behave differently based on the time of the day, to handle busy and calm hours. It could take into account the amount of flows currently allocated by the requester, or the total bandwidth used, or any other information known to the network.

4. Modern policy-based networking on your Linux

rlite is one of several existing RINA implementations. It is an open-source project started by Vincenzo Maffione, licensed under LGPLv2, with the goal of creating a simple, performant and production-ready implementation. It targets GNU/Linux systems as the platform, as some parts exist as kernel modules.

In its current sate, rlite supports all the functionality required for basic network communication, such as:

- Arbitrary stacking of DIFs
- Dynamic DIF enrollment
- Flow and retransmission control
- Inspection tools to query status information
- Implementation of the CDAP protocol for application communication
- Ability to run over legacy media like Ethernet, WiFi or UDP

Aside from the networking stack itself, rlite also includes a POSIX-like API similar to the socket API to facilitate the transition for programmers and help ease the process of porting existing code.

Some applications have been ported to use the rlite API, such as the Dropbear ssh server and Nginx web server, and are available in the project's GitHub. Additionally, tools to interoperate with IP networks have been created, such as a gateway between rlite and IP networks, as well as a tool to tunnel IP traffic through rlite.

A performance evaluation can be found in the project's README. On two consumer hardware Linux computers connected by a 40Gbps link, it has been able to reach speeds of up to 10Gbps for reliable traffic.

As it was described in the previous chapter, policies are very important for RINA and a cornerstone for a lot of its functionality. Most of my contributions have been focused on bringing policy functionality to rlite. Additionally, I have implemented support for a WiFi shim, allowing rlite to support DIFs that use wireless links.

According to GitHub, this resulted in almost 9000 lines of code changed, with about 2000 being added. Before my improvements, it was not possible to change the behaviour of the various components, nor extend the functionality with optional policies in a unified way.

This required abstracting out the components and implementing their default behaviour as a policy. As rlite is implemented in C++, all policy classes need to

---

[11] Which are the same thing.

inherit from the base class and implement the necessary virtual methods. A framework exists for registering available policies, and the control tool for managing the stack was extended to allow changing the (DIF) policy at runtime. Policies can also have configurable parameters that are specific to a particular instance of that policy. These can also be adjusted at runtime using the aforementioned tool. Safety checks are in place to ensure that the choice of policies and parameters makes sense within the environment of the DIF.

Policies can have dependencies between themselves. Therefore, a system was introduced to allow declaring these in a trivial manner. An optional list of dependencies can be supplied when registering the policy. Additionally, a group registration call exists, which automatically creates cyclical dependencies among all members of the group. There is a dependency resolver in place, which builds and traverses the dependency tree of the requested policy. It checks whether all declared dependencies are known, and whether there are no conflicts (that is, multiple policies belonging to the same component). In case no errors occur, all the policies are switched at once, in reverse order (the requested policy is activated last).

Switching policies therefore behaves in a transaction-like manner, ensuring that the DIF does not go into an incoherent state if an error was encountered after switching e.g. half of the policies. This is especially useful for security, as some (block) ciphers might require the packet fragmentation to be done at certain boundaries, etc., but the use is not limited to just that. Bundling policies as dependent and interconnected sets allows greater freedom for implementation, as it is now possible for the policy to assume some behaviour of another component, while having these relationships be strictly enforced at the system level.

Research efforts greatly benefit from policy support, as it allows rapid and easy experimentation with the network. New ideas, e.g. multipath routing algorithms, can be easily implemented and later switched between on the fly, allowing to observe and measure the behaviour under the same circumstances. There is no need to reboot the machines or recompile rlite and distribute the updates to all network nodes when switching policies.

Most security features are a matter of policy, such as authentication when joining a DIF or traffic encryption. [13, 14] As security is paramount for real-world deployments, providing an easy to use framework for these is crucial for encouraging third parties to consider RINA as a viable option for their infrastructure.

Let us illustrate with a concrete example, by showcasing how to create a new routing policy. The basis for the routing component is the `Routing` class, describing the methods that are necessary to be implemented. We will therefore inherit from this class for our custom routing policy. (See figure 5).

```
class MyPolicyClass : public Routing {
  MyPolicyClass(UipcpRib *rib) : Routing(
      rib);
  ~MyPolicyClass();

public:
  void dump_routing(std::stringstream &ss);

  void update_local(const std::string &
      neigh_name);
  void update_kernel(bool force = true);
  int flow_state_update(struct
      rl_kmsg_flow_state *upd);
  void neighbor_updated(const std::string &
      neigh_name);

  void neigh_disconnected(const std::string
       &neigh_name);

  int route_mod(const struct
      rl_cmsg_ipcp_route_mod *req);
}
```

Figure 5. The class declaration of our custom routing policy, listing the methods that need to be implemented.

Once implemented, we need to register the policy, using the `UipcpRib::policy_register` method. (See figure 6).

```
UipcpRib::policy_register(Routing::Prefix,
    "my-policy-name",
    [](UipcpRib *rib) {
        return utils::make_unique<
            MyPolicyClass>(rib);
    },
    {Routing::TableName});
```

Figure 6. Registering the custom policy.

Afterwards, we can switch the routing policy by running

```
rlite-ctl dif-policy-mod \
  <DIF-NAME> routing my-policy-name
```

in the shell.

5. Conclusions

In this paper, we have explored some of the flaws of traditional TCP/IP based networking, outlined how the Recursive InterNetwork Architecture works, and shown how rlite implements policies and why they are beneficial for both research and production environments.

As it stands right now, there exist structural deficiencies that cannot be solved properly merely by introducing new protocols or abstractions into the protocol stack. Although layers were the proper model, the meaning of layers and their purpose was misunderstood. *Layers are distinguished by the scope of their shared state, not their functionality.* Research into alternative architectures is therefore worthwhile and necessary.

Dynamic policy switching is supported by rlite and a number of policies are included by default. While none of them have any dependencies, it is possible to declare such relations. The framework provides a transaction-like behaviour to ensure that the system never goes into an incoherent state. All of this is covered by unit tests which are part of continuous integration.

rlite is being continually improved. For my master's thesis, I aim to leverage the policy support to explore flow allocation approaches that take bandwidth requirements into consideration. Application requesting a flow includes information about minimal required bandwidth, and the request is rejected if a path cannot be found in the network (DIF) graph that would satisfy these demands. If such a path is found, bandwidth is reserved along the path so that the application is guaranteed to have this bandwidth available at all times, thus avoiding congestion.

The OCARINA research group at University of Oslo has shown interest in using rlite and its policy support in their research efforts into routing and congestion control. Some currently present features, such as the `configen` tool, have been contributed to help fit rlite into their environment during my study exchange there.

Acknowledgements

I would like to thank my supervisor Vladimír Veselý for his help with writing this paper. Additional thanks belong to my friend MarsCat for giving me continuous feedback on the style and direction of this work, as well as help with designing the unofficial rlite logo.

References

[1] John D. Day. *Patterns In Network Architecture: A Return to Fundamentals*. Prentice Hall, 2010. ISBN: 0-13-706338-5.

[2] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 1883, December 1995.

[3] Google. IPv6 – Google. online (english). `https://www.google.com/intl/en/ipv6/statistics.html`.

[4] Dino Farinacci, Vince Fuller, David Meyer, and Darrel Lewis. The Locator/ID Separation Protocol (LISP). RFC 6830, January 2013.

[5] Vladimír Veselý. *A New Dawn of Naming, Addressing and Routing on the Internet*. PhD thesis, Brno University of Technology, Faculty of Information Technology, 2016.

[6] Ross Callon. The Twelve Networking Truths. RFC 1925, April 1996.

[7] ISO/IEC 7498-1:1994. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. Standard, International Organization for Standardization, November 1994.

[8] Andrew S. Tanenbaum and David J. Wetherhall. *Computer Networks*. Prentice Hall, 2010. ISBN: 0-13-212695-8.

[9] Robert T. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, October 1989.

[10] Robert T. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123, October 1989.

[11] John Day. The Interina Reference Model, 2014. Pouzin Society.

[12] Pouzin Society. Terminology — Pouzin society. online (english). `http://pouzinsociety.org/education/terminology`.

[13] E. Grasa, O. Ryšavý, O. Lichtner, H. Asgari, J. Day, and L. Chitkushev. From protecting protocols to layers: Designing, implementing and experimenting with security policies in RINA. In *26th IEEE Internationl Conference on Communications (ICC)*, pages 1–7, May 2016.

[14] H. Asgari. Deliverable 4.1: Draft conceptual and high-level engineering design of innovative security and reliability enablers. Technical report, Programmability In RINA for European supremacy of virTualised nETworks, September 2014.