MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Network Manager for RoFI platform

Master's Thesis

VLADIMÍR CHLUP

Brno, Fall 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Network Manager for RoFI platform

Master's Thesis

VLADIMÍR CHLUP

Advisor: Prof. RNDr. Jiří Barnat, Ph.D.

Department of Computer Science

Brno, Fall 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vladimír Chlup

Advisor: Prof. RNDr. Jiří Barnat, Ph.D.

Acknowledgements

I want to thank my wife, Kate, for her unending support and love.

Abstract

This thesis focuses on designing and implementing the Network Manager for the RoFI platform. A software tool for managing and controlling the networking stack allows users to configure individual network interfaces and design and implement their network protocols. The Network Manager improves the overall routing support of the platform, provides two additional routing protocols, and expands the extensibility of the whole networking stack. A part of the work is a demo application and an experimental verification of the implemented solution.

Keywords

routing, RoFI, IPv6, routing protocol, networking, robots

Contents

1 Introduction **1**

2 State-of-the-Art **3**
2.1 *Networking model* . . . 3
2.1.1 Single module – the universal module . . . 3
2.1.2 Networking architecture . . . 4
2.2 *Routing* . . . 5
2.2.1 Routing table . . . 5
2.2.2 Routing protocol . . . 6
2.2.3 Implementation . . . 7
2.3 *Discussion* . . . 8

3 Network Manager **9**
3.1 *Network management tools* . . . 9
3.2 *Responsibilities* . . . 10
3.3 *Model* . . . 11
3.4 *Interfaces* . . . 13
3.5 *Routing Table* . . . 14
3.6 *Protocols* . . . 14
3.7 *Network Manager logic* . . . 15
3.7.1 Adding and removing addresses . . . 16
3.7.2 Setting the state of an interface . . . 17
3.7.3 Passing messages . . . 17
3.7.4 Setting protocols on interfaces . . . 17
3.7.5 Handling protocol update messages . . . 17
3.7.6 Connecting and disconnecting connectors . . . 19
3.7.7 Static routes . . . 19
3.7.8 Summary . . . 19
3.8 *Discussion* . . . 20

4 Implementation **21**
4.1 *Network Manager* . . . 21
4.1.1 Managing interfaces . . . 22
4.1.2 Managing protocols . . . 23
4.1.3 Managing routing table . . . 24
4.2 *Interface* . . . 24

4.3 *Routing Table* . . . 26
4.4 *Protocol* . . . 27
4.5 *The main logic* . . . 30
4.6 *Logging* . . . 31
4.7 *Command Line Interface* . . . 31

5 Protocols **33**
5.1 *Leader Elect* . . . 33
5.2 *Simple Periodic* . . . 35
5.3 *Simple Reactive* . . . 37
5.4 *RRP – RoFI Routing Protocol* . . . 39
5.5 *Summary* . . . 42

6 Experimental evaluation **43**
6.1 *Demo applications* . . . 43
6.1.1 Communication example . . . 43
6.1.2 The Network Manager CLI application . . . 46
6.2 *Experiments – protocols* . . . 47
6.2.1 Various networks . . . 47
6.2.2 Leader Elect . . . 50
6.2.3 Routing protocols . . . 51
6.3 *Discussion and Limitations* . . . 56

7 Conclusion **58**
7.1 *Future work* . . . 58

Bibliography **60**

1 Introduction

The RoFI platform is a metamorphic robotic platform developed under the ParaDiSe laboratory of the Faculty of informatics at MU, first introduced in the master thesis of Jan Mrázek [1].

The metamorphic robots – RoFIbots – are constituted of independent, self-sufficient modules. These modules have their own computing power, battery and limited ability to move. However, these individual modules are able to connect to each other physically and, thus, can form more sophisticated robots with different shapes and forms to adapt better to the current task. Therefore, the ability to have reliable and fast communication is crucial.

In the previous, bachelor's thesis [2], we designed the communication model of a single rofibot and implemented routing support into the platform. Although the current solution is functional, it has some drawbacks, such as missing user-space control, lacking general network protocols support or missing extendibility.

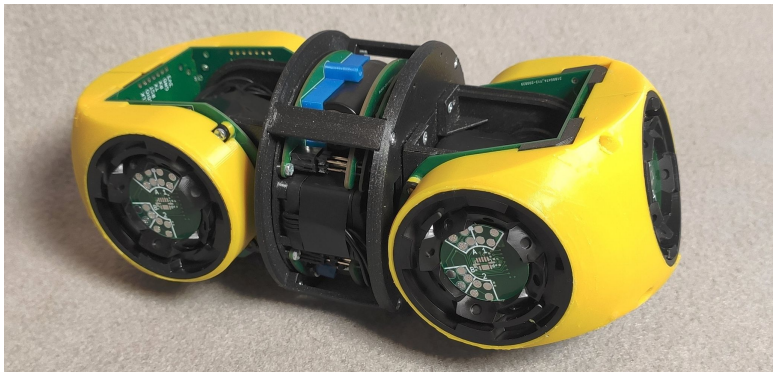

Figure 1.1: A universal module – the platform's building block [3].

In this thesis, we are going to address some of those drawbacks and introduce a network manager for the RoFI platform aiming at better user control of the networking capabilities of RoFIbots, including a specification of the API for various network protocol support.

Chapter 2 presents the current networking solution as introduced in the bachelor's thesis. We summarize its design, implementation, its features and disadvantages.

In chapter 3, we describe the network manager itself as a concept and a tool, along with its purpose and responsibilities, with an emphasis on the previous chapter. We discuss what of the current solution may remain and which parts need to be changed.

In chapter 4, we introduce the actual network manager's implementation. We describe its overall architecture and design, including the application interface of the network manager and its components. We also describe an additional component for the network manager – a command line interface (CLI) – which has proven itself useful for interacting with the proposed solution from the user space while an application is running.

Chapter 5 is dedicated to one of the main components of the network manager – protocols. This chapter provides concrete implementations of various protocols by which the solution's functionality can be easily extended. The chapter contains multiple protocols with a simple functionality yet using different features of the network manager, such as routing protocols with both periodic and reactive update messages and protocols affecting the state of the manager's interfaces only – this is demonstrated on a simple leader-elect algorithm.

In chapter 6, we focus on experiments and simple proof-of-concept applications built using the described and implemented solution. We also discuss the results and possible limitations.

The last chapter, chapter 7, concludes the thesis with a summary of the work and a few possible routes and areas for future work are proposed.

2 State-of-the-Art

The current networking solution was introduced in a bachelor's thesis from 2020 [2] that aimed to design and implement routing support into the RoFI platform. The solution build on the TCP/IP stack provided by the lwIP library [4]. However, the library is oriented to end-point devices and thus absent of the routing support by default.

The previous work consists of two main parts; the first one is oriented to the whole networking model of the RoFI platform, which was redesigned in the thesis; the second one is the design and implementation of the routing.

Although the output of the bachelor's thesis was a functional networking stack, it has certain flaws we aim to address in this work.

This chapter is divided into several parts. The first part summarizes the networking model itself, and the second presents the actual routing within the platform. In the third part, we discuss the advantages and disadvantages of the described solution, and we argue which of its components can be preserved and which have to be adapted to meet the new demands.

2.1 Networking model

The networking model from 2020 is itself a redesign of the original solution. It defined a scheme of intra-module communication as the basic built block of the whole network, which is only a set of such interconnected modules. To better understand the architecture, we will describe the networking design on the universal module itself with some of its main key features.

2.1.1 Single module – the universal module

The universal module is the building block of every RoFIbot. From the networking perspective, it is a device with multiple physical connectors called RoFICoM over which traffic can flow. The universal module has six RoFICoMs (although the current solution is general enough so that it can work with different modules with a varying number of connectors – however, at the time of writing this thesis, no other

modules are yet available) and is able of sending and receiving data from directly connected connectors of neighbouring modules.

Figure 2.1: A universal module mounted on a pad [3].

These connectors do not process traffic only. They also provide the ability to detect and react to connection changes. That means that the network stack can detect if an active connector gets disconnected from its counterpart or, conversely, if a new connection is established when a new module connects. This is a feature that is utilized later.

2.1.2 Networking architecture

The networking architecture consists of six networking interfaces corresponding to the physical connectors – RoFICoMs. Each of these interfaces has its IP addresses – one of the significant changes of the redesign was the transition from IPv4 protocol to IPv6 protocol, which allows us to have multiple addresses on a single interface, a feature used later.

The fact that these networking interfaces correspond to a particular RoFICoM means, that their input and output is well defined. When the interface wants to communicate, the data must flow through the

connector. For the networking, the RoFICoM can be thought of as a network card with a firmware, and the networking interface is then a driver with a further functionality.

In addition to these interfaces, there is one extra – so-called virtual – interface which is intended to represent the main interface of the module. This approach has several advantages.

One of them is that the global IP address can be assigned to this interface and, therefore, to the whole module without being linked to any physical interface directly. This means that this interface can exchange traffic with all interfaces equally – such a message is processes internally within the networking stack and then the appropriate physical interface (i.e., interface with underlying RoFICoM).

Another advantage is that this interface is always reachable as long as there exists a path for traffic (i.e., there exists at least one working connector) – this is something we can usually encounter in networks, sometimes under the concept of routing to the loopback interface. In case we would not have such a virtual interface in the model, we would have to assign the global address to one of the six physical interfaces, which would cause two potential problems: first, we would artificially prefer one of the otherwise equivalent interfaces; second, this *special* interface could not go ever down; in such case, the module would become unreachable even in case there would be another interface that would be reachable.

2.2 Routing

Once we have established the networking model currently implemented within the platform, we focus now on the routing process. The routing consists of two main parts – a routing table and a routing protocol.

2.2.1 Routing table

A routing table is a (module-wise) global database of routes known to the module. Every time a message is solicited to be sent from one module to another, the source module has to consult the routing table and choose an interface through which the message should be sent.

Then the next module repeats the process until the message either reaches the destination module and is processed or arrives at a module which does not have a corresponding route in its routing table and then the message is dropped and does not propagate any longer.

To decide correctly which interface should be chosen for a given message, the routing table has to hold enough information based on which it can decide (based on the message's destination address) which interface to use as a gateway – an output interface.

The platform's table keeps records for individual networks – a network address and its subnet mask – and for each of these networks, it contains a list of individual gateways to this network with their cost. The gateway from corresponding records with the best cost is chosen for each message into the particular network.

As a matter of fact, the current networking solution does not use the routing table directly. There is an additional table – called the forwarding table – that contains the best paths for each network only. Thus, the table is smaller and easier and more efficient to go through; a disadvantage is that the routing table has to secure that the data in the forwarding table are always kept up to date. Otherwise, the routing would not work properly. This is more or less an implementation detail which we will use in chapter 4.

2.2.2 Routing protocol

A routing protocol is a protocol which ensures that routing tables of all individual modules are correctly populated and safeguards that information about routes and their changes are shared properly across the network. It also specifies the format of the update messages (i.e., messages containing information about routes and their changes) and the way and time they are shared.

In the current solution, there is only one routing protocol. It is called RRP (from RoFI Routing Protocol), and it uses various features, we mentioned in the previous part of the chapter (such as the ability to react upon connection changes or the ability to have a single interface with multiple addresses.

The RRP introduced a relatively complicated scheme of message exchange with an aim of limiting the number of messages to be shared between modules. It distinguishes multiple message types (e.g., *Hello*

and *HelloResponse* for new connections, *Call* and *Response* for existing connections), and its messages contain only the changes that took place from the time the last message was exchanged.

Another feature of RRP is that the messages are sent on network change only. In one of the previous parts of the chapter, we mentioned that the connectors of the modules can react to on connection and disconnect of a counterpart module. The RRP takes advantage of this ability, and besides the initial *Hello* message that is sent during the protocol initialization, the only time a routing update is sent is when the topology of the network changes.

The protocol update messages are commonly shared via the multicast; each network interface with the underlying connector listens on a multicast address defined by RRP, and onto the same address, it sends its potential updates as well. This solution relies on the fact that multicast addresses can be listened to by multiple devices, and it does not require any additional pre-agreement between the individual modules.

2.2.3 Implementation

We have described the architecture of the current solution. We will now focus on the actual implementation.

The networking is implemented using C++ programming language under an umbrella class *RoIF* which includes the routing table and network interfaces. It serves as a proxy for the virtual network interface and therefore is used in case the user wants to add or remove the module's global address. The initialization of the class means the initialization of the module networking.

The class also handles passing incoming and outgoing routing protocol messages between individual interfaces, and the routing protocol and the table. The protocol is implemented within the routing table.

2.3 Discussion

We described the current solution as introduced in the bachelor's thesis. Although the solution is usable, it has some flaws worth addressing.

The main flaw is the single routing protocol available without any possibility of easy extensibility as the code of the routing protocol itself is heavily interlinked with the code of the routing table. This proves to be a problem due to various reasons.

Besides the decrease in approachability, the potential user has to think about two functionalities at once; the first one is that the RRP is, in some cases, a rather unnecessarily complicated protocol, which make the routing stack of the platform needlessly less friendly.

The second flaw is that the RoFI platform is a universal platform and its possible use cases are too diverse to expect that one solution will fit them all – and even if it would, we see no reason to deny the user the possibility of using potentially worse solution if they wished to. The third flaw is the fact that this artificial restriction limits many applications of the RoFI platform, which might serve as, for example, a prototyping platform to try and test a plethora of routing protocols for application in robotics and resource-limited networks.

Another issue is the *RoIF* class itself due to the fact that it serves both as a proxy for the virtual interface and for the complete networking functionality. This results not only in unnecessary code duplication between its implementation and one of physical interfaces but also in reduced functionality – the current ability to configure the physical interfaces is technically non-existing; although the various functionality is present, the user has no access to it. The virtual interface in the current implementation is also, in some sense, a *special* interface, although the main difference is only in the missing physical connector underneath it – this fact should not, however, obscure that the rest of the functionality is the same and can be treated as such.

We will address the above in the implementation of the Network Manager, which can be found in chapter 4. Before that, we will specify the notion of the network manager as a software tool in the next chapter, including its responsibilities and desired capabilities.

3 Network Manager

The key term of this thesis is a network manager. Therefore we try – before running into specifics of the actual solution – to approach this subject from more common point of view – user-space of the current operating systems such as MS Windows or Linux distributions, which the reader probably knows well from the user-perspective. Then, we will pass to the network administrator point of view. In the end, we will abstract some of the common functionality and describe the proposed network manager for RoFI platform.

3.1 Network management tools

The network manager is a software tool that manages network connections and network interfaces within the system. It can also gather various information about configuration changes and potential communication issues such as mismatching configuration, etc.

In Windows, the network settings are part of the built-in settings tool. There are usually listed all available interfaces; left-clicking reveals more detailed information about the individual interface. For each interface, the user can configure its IP address, whether it will be given by a DHCP server, or whether it will be entered by the user manually. In the latter case, the user has to enter not only the address, but also network's mask and the default gateway (that is usually the router).

Windows has also a more advanced tool than the one integrated within settings – Connection Manager. This tool gives user a greater control over the hardware. It can enable and disable individual interfaces and configure them, even update corresponding drivers.

In world of Linux, there are many tools for the network control and management. We will mention only one of them with a convenient name – NetworkManager [5]. This tool is widely used across many distributions such as Ubuntu, Fedora, or Arch Linux. It also provides an ability to configure network interfaces, their addresses and state. It can also set up static routes. The graphical user interfaces (GUI) of Linux distributions have usually a built-in support for the NetworkManager so that it can be configured from the GUI using a mouse or

touch too. However, the tool itself provides a command line interfaces as well (called *nmcli* and *nmtui*).

These tools, both Connection manager and NetworkManager, are more or less oriented towards the end-point devices; e.g., desktop computers. However, the universal module (and potentially other module types in the future) has a role of a router within the RoFIbot's network as it has to decide, upon receiving a message, if it is destined for itself, or if it is intended for some other module within the network to which the message should be forwarded, and if so, then which interface (RoFICoM) shoud be used. All of this means, that there is a functionality we do not encounter often from the point of view of a user of desktop operating systems. Fortunately, we can inspire ourselves with the networking industry.

Cisco is a big networking vendor which makes its own routers and other network devices running their Internetwork Operating System (IOS) providing implementation of interface management, support for routing protocols, VLANs, network protocols in general and more. User, or rather more precisely administrator/network engineer, can control and configure the device. As specialized devices, routers provide much more configuration options and much more built-in area-related functionality.

The RoFI Network Manager should support both approaches. The first approach are configuration options, that are more straightforward and simplier for platform's users who want to use networking, but are only interested in the ability of sending messages between different modules (e.g., those users are not interested in the chosen routing protocol). The second approach is oriented towards those users, who are building some lower-level algorithms themselves and therefore prefer more granular and advanced configuration options. They may need a better grasp and better control over the networking within the RoFIbot.

3.2 Responsibilities

First of all, we specify the main responsibilities of the Network Manager tool we are proposing. We have seen two different approaches towards the networking, but those views varied in the amount of op-

tions and choices available to their users. Now, we want to distinguish areas over which the network manager will have control – something which overlapped even with the two different approaches above.

One of the main tasks of the Network Manager is the control of network interfaces. This control includes the state of the interface – it can be either up, and therefore processing incoming and outcoming traffic and reacting to it, or it can be down, and in that case no traffic is processed. An interface can also have multiple addresses which can be added, changed or removed in real-time. In addition, there can be also (potentially multiple) protocols running.

Another task of our Network Manager is protocol management. Network Manager should support multiple protocols, that can be then running in the system, and/or on particular network interfaces at the same time. It should permit the user to set protocols up and down in real-time while the system is running and also to inform him about their state and resources. These protocols provide us with a way of communication between individual modules and therefore modules can communicate and even influence the system by informing it about existing paths and thus affecting a routing table.

The last task and the last component which will the Network Manager be responsible for is the aforementioned routing table. Routing table is a system-wide table (i.e., one routing table for the whole module) containing information about known routes within the RoFIbot's network. When we want to send a message from one module to another, we have to consult the routing table to determine which of the six connectors (in case of the universal module) we must use.

3.3 Model

Now, when we established the main responsibilities of the Network Manager, we will describe the architecture and design of the proposed solution. In this chapter, we will do it by general description of the overall concept without diving into unnecessary implementation details. We dedicate the chapter 4 to the implementation.

The Network Manager will control all three areas we had described above: interfaces, protocols, and routing table. In addition to the management of those components, it has to provide necessary logic be-

tween them; e.g., passing messages between interfaces (where messages are received using the underlying RoFICoMs) and corresponding protocol (where messages are processed), or applying changes between interfaces and routing table after an address is added to or removed from an interface. The network manager has to provide also an interface for users – a way to add an address to a network interface, etc.

Before we describe the architecture in more detail, we will specify the individual components first. However, to attain a bigger picture, there is a scheme 3.1 describing various actions which can take place and respective components it affects.

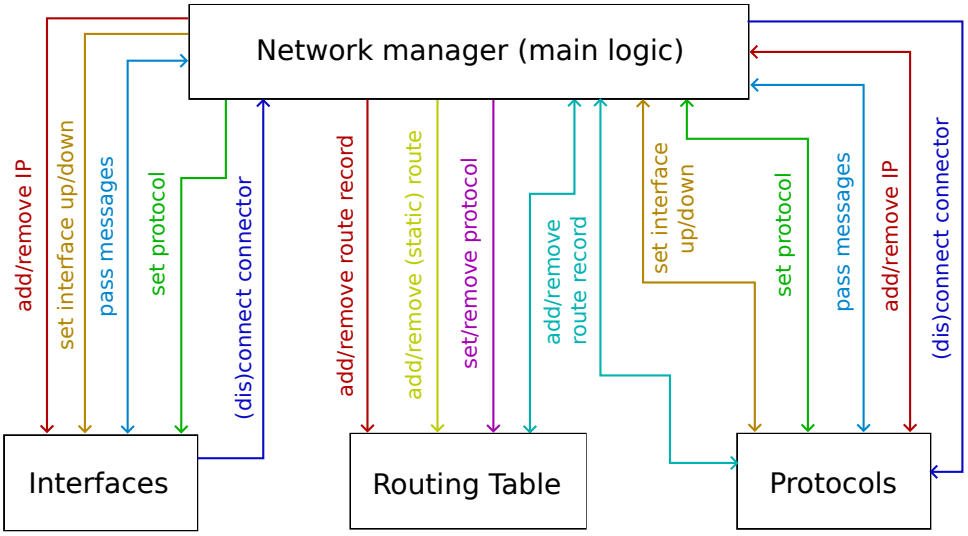

Figure 3.1: An outline of the Network Manager's main logic.

The same or similar actions that are somehow connected have arrows of the same colour; for example., the most left arrow has a red colour and indicates an addition or a removal of an address on an interface, which is an action that originates in the Network Manager and influences the interface. However, the interface itself is not the only component affected, as there is a routing table, that might contain the old address (in case of the removal) or be missing the new one (in case of the addition), and therefore must be updated. There might be a protocol affected too, as the protocol manages the interface and shares its IP accross the network. In that case, it has to inform other

agents in the network about such a change. An attentive reader would notice, that the arrow between the protocols and Network Manager points in both directions. This is due to the fact that the protocol itself can cause change of an address of an interface, so it can be protocol itself who informs the network manager, not the user only.

3.4 Interfaces

The first component we focus on is the interface. The overall interface layout of an individual module remains similar to the current model. We consider one network interface for each physical connector with one additional virtual interface. The main changes are in the unification of the interfaces themselves and added advanced configurability.

A network interface has usually a unique name within the device. It has also potentially multiple addresses – we keep using IPv6 addressing – that should be unique within the network, and new can be added or removed. It has also a state as we have already mentioned; it can be either up or down.

The state of the interface has an influence on a traffic. The interface can receive data and it can also send data – both require the interface to be up. It has to support multicast; i.e., listening to multicast addresses onto which the interface is subscribed.

Handling traffic is the only aspect where the physical interface differs from the virtual one. The physical interface receives the incoming messages from the underlying physical RoFICoM, the data are sent by the connector conversely. The virtual interface does not have such a connector. It has to, therefore, rely on the routing (i.e., the content of the routing table) when deciding which connector to choose.

This is not a problem if we consider that the module itself is a router. This means that the physical connector can output data from an interface in two cases – the first is the local link communication between two neighbouring modules where no routing is needed in the same way no routing is need in case two computers are connected through an ethernet. The second case is when the communication happens with an arbitrary agent within the network, in which case the routing table is equally needed.

The virtual interface works the same way except it does not require any (physical) link for communication as everything takes place within the module's TCP/IP stack provided by the underlying library.

3.5 Routing Table

The second component we presented was the routing table. The routing table has to contain all routes known to the module, therefore it has to provide the ability not only of adding and removing routes, but also the ability of searching among the records present.

Individual records have to correspond to unique networks – an IP address with a mask[1] – and they have to comprise all routes to a given network. This means, that each record can itself contain multiple routes which can differ in a gateway (i.e., the output interface) and/or in a cost. Furthermore, they may differ in their origin as well, but we will focus on this in a short while.

Cost is a metric that can be computed in many ways, however, the lower cost the better[2] the given route ought to be. The cost may even depend on the protocol which introduced the given route.

One of the aims of this thesis is to extend the RoFI platform to support multiple protocols. This means, that in the routing table there might be many routes from many protocols (even similar otherwise), therefore, for each route, we will hold information about the protocol that introduced it. However, it is worth pointing out, that not all routes have to be discovered via protocol, we still want to support user-entered static routes.

3.6 Protocols

The last component are protocols. As described in the previous chapter, the current solution includes only one protocol – the platform-specific routing protocol RRP. We want to introduce an interface describing

1. The term *prefix* is commonly used.
2. However, the computation of the cost defined what the *better* mean. Sometimes, the shorter path is preferred, but there exist situtations where longer paths might be preferred (e.g., routing across autonomous systems where some shorter route usage may be linked to extra fees). This is however out of scope of this thesis.

capabilities of the routing protocols in general. However, we will not limit ourselves to routing protocols only, we propose an interface that supports two types of protocols; the routing protocols that introduce new routes or remove the old ones, and protocols that influence the configuration of the modules – this can be used, for example, for a simple leader elect protocol as we show in chapter 4 or for protocols such as the Spanning Tree Protocol (STP) which can be used on network switches to prevent loops within the network by disabling certain interfaces. The most of the functionality will be shared between the two types.

The two main attributes the protocol within the platform has to have are the name, and the multicast address on which the protocol shares its update messages.

The protocol has to be able to create and process the update messages and then report possible changes to the Network Manager. This reporting of changes is in fact one of the two ways the interface of the routing protocol will differ from the non-routing one.

The protocol has to be able to tell which interfaces it manages (i.e., on which interface it runs), and therefore which interfaces can take place in the protocol's communication process (such as sending updates, etc.). Other interfaces are not taken into consideration. Also, we have to be able to add a new interface to the protocol and, conversely, to remove managed interfaces too. In addition, the protocol should provide a way how the protocol itself can be notified about the change in the state of an interface as we discussed earlier in the figure 3.1.

Additionally, the routing protocol has to be able to access the routing table of the module, or more precisely, has be able to access routes discovered by itself. Alternatively, the protocol could cache all the information it passed down the routing table by itself, however, this functionality would be often duplicated between the actual implementation of routing protocols, and it is more prone to errors than the single, general solution.

3.7 Network Manager logic

We described the key components of the Network Manager. Now, we can focus on the main logic of the manager, that has to properly connect

the individual components and secure proper data flow between them as outlined in figure 3.1 which we repeat below. We will now look more closely on the individual actions.

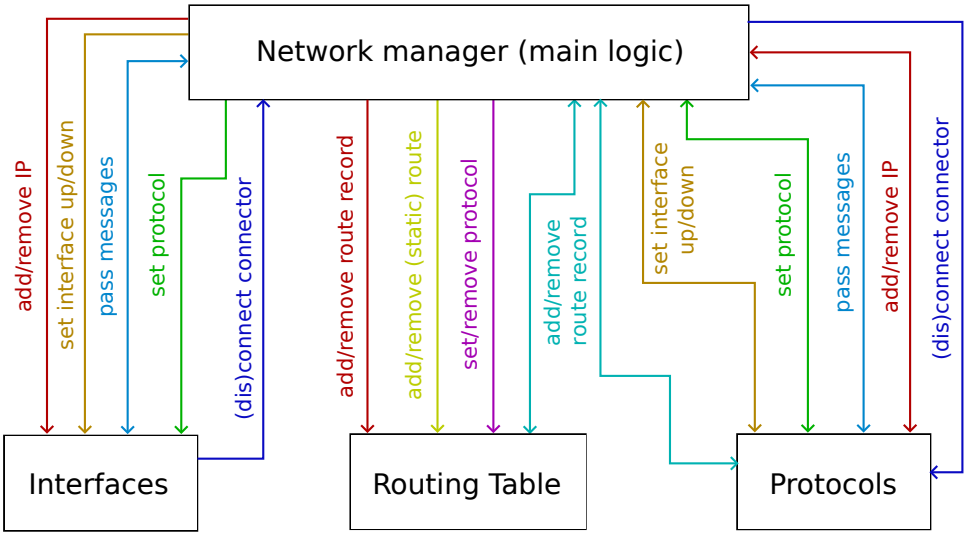

Figure 3.1: An outline of the main logic. (*Repeated figure*)

3.7.1 Adding and removing addresses

The first action on the figure (from the left) is adding and removing of an address from the interface. This action can arise from two sources; the first is the user action, when an address is manually added or removed, the second is the configuration update obtained through some protocol. The Network Manager's response is in both cases similar. First, it has to add or remove the address from the interface. Then, the manager has to propagate the potential changes to the routing table; i.e., it has to add or remove corresponding route. Finally, all protocols managing given interface have to be informed about all the changes that took place (if any), which might trigger the whole process anew. That would be then handled as a new action.

3.7.2 Setting the state of an interface

The second action is the setting interface up and down. This is one step simplier than the action before, because it does not involve the routing table directly. The Network Manager has to inform all protocols that are affected by the change. If any update to the routing table or any interface is required, the protocol will provide an appropritate update as in the previous action.

3.7.3 Passing messages

The Network Manager has to pass messages between the two components; interfaces and protocols. Interfaces are the component that is linked to the physical connectors, so every message goes through it. The Network Manager has to handle the messages destined for individual protocols and pass them to it accordingly. The protocol then has to be able to send messages too, because after the protocol receives a message, it might need to send a response[3]. This has to be secured by the manager.

3.7.4 Setting protocols on interfaces

When we discussed interfaces and protocols, we mentioned that the protocol can be set to run on an interface and therefore manage the interface. When the user sets up a protocol on a given interface, the Network Manager has to set up a listener on the protocol's multicast address and notify the protocol to manage the interface. In case of a removal of an interface from the protocol management, the manager has to remove the listener and properly clean the possible records from the routing table.

3.7.5 Handling protocol update messages

The handling of protocol messages is a complicated process, therefore, it might be useful to describe it in detail. The whole procedure is captured in the figure 3.2.

3. Naturally, the update message can result in need to update known routes, etc., but this is handled as a separate action.

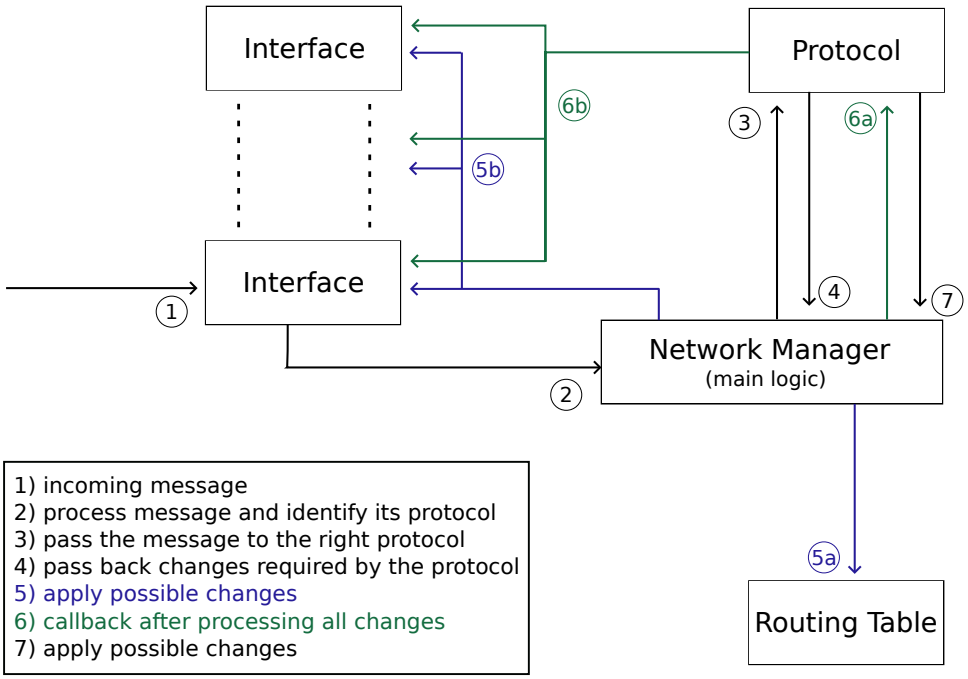

Figure 3.2: The scheme of handling protocol messages.

It starts with the incoming message (1). The message is addressed to a certain multicast address, based on which the destined protocol is chosen (2) and the message is passed to the corresponding protocol (3). The protocol processes the message and passes potential changes to the Network Manager (4). If no updates are passed, the process ends. Otherwise, the Network Manager processes the changes which can be of two types; the updates may involve the routing table (5a) in case the protocol is a routing protocol and new routes were discovered or some old ones disappeared, or the updates may request some configuration changes of certain interfaces (5b) in case the protocol is not the routing protocol. After those changes are processed by the Network Manager, the protocol is notified and it can update its internal state (6a) and also send a message through all of its managed interfaces (6b) – this is useful when, for example, the routing update arrived onto one interface resulting in the new route discovery; then it is necessary to inform the other neighbouring modules within the network so that the route is propagated. Finally, the protocol may once again pass

possible changes to the Network Manager (7). Once this process is completed, the update message was processed.

3.7.6 Connecting and disconnecting connectors

The non-virtual interfaces have an underlying physical connector over which the data flow. As we discussed earlier, the platform provides tools for detection of new connections and disconnections of the connector. In case the interface is in use (e.g., it has an address, is managed by some protocol and data are passing through), the disconnection has ramifications, and the Network Manager has to inform all associate protocols. This will probably lead to some other changes which can then trigger appropritate actions (namely passing messages to other modules about the changes in the network topology, and the routing table updates). There might an interface that is managed but disconnected, suddenly connecting to another module, then all affected protocols have to be informed too and the changes requested upon such action must be processed.

3.7.7 Static routes

When we described the new solution, we mentioned the user should have an opportunity to introduce static routes into the networking. This is also handled via the Network Manager which passes such a request to the routing table. The user is required to enter a network (i.e., an IPv6 address and a subnet mask), the gateway interface and the cost of the given route. Static routes can be removed in a similar manner.

3.7.8 Summary

We have seen the individual actions that must be handled by the Network Manager's logic interconnecting the individual components of the model.

3.8 Discussion

We have described the new Network Manager tool. In comparison with the previous chapter, we have seen some significant changes related to the current solution (i.e., the separation of the routing table and a protocol, or even the support of multiple protocols at once), but also only a small evolution in case of the actual network interfaces and the design of the whole networking model which remained basically preserved (the count of interfaces, the virtual interface, a connectivity between them, etc.).

In the next chapter, we will describe the implementation of the Network Manager.

4 Implementation

This chapter describes the actual implementation of the Network Manager and it is divided into two parts. The first part is dedicated to the implementation of the Network Manager itself as was described in the previous chapter including its components; interfaces, routing table, and the general interface of the protocol. The second part is dedicated to the Command Line Interface (CLI) for the Network Manager that aims at those use cases where it is desired to control the networking interactively from the user space without forcing users to implement their own application (although it is still possible).

Even though this chapter focuses on the actual implementation, it contains only the core functionality of the manager. It means that this chapter does not contain an implementation of any protocol, it focuses only on the general protocol interface. The concrete protocols and their corresponding implementations are the subject of the next chapter.

The Network Manager aims to replace the current networking within the RoFI platform. The platform's base code is written mainly in C++ programming language. The Network Manager will be no different. Written in C++, the Network Manager will take place as a part of the RoFILib (a set of the platform's libraries independent of the underlying platform's implementation). These implementations can be either physical modules, or a simulation of those modules using one of platform's simulators. The Network Manager supports both and is implemented within its `rofi::net` namaspace.

4.1 Network Manager

The Network Manager is represented by the class `NetworkManager`. Its constructor (i.e., function that returns a new instance of a given class) requires the RoFI proxy – that is a platform's proxy for underlying module. It contains all necessary information about the underlying module's characteristics such as the number of connectors or the module's unique ID. Optionally, the constructor accepts a MAC address. If no MAC address is supplied, the ID is used to construct the MAC address arbitrarily.

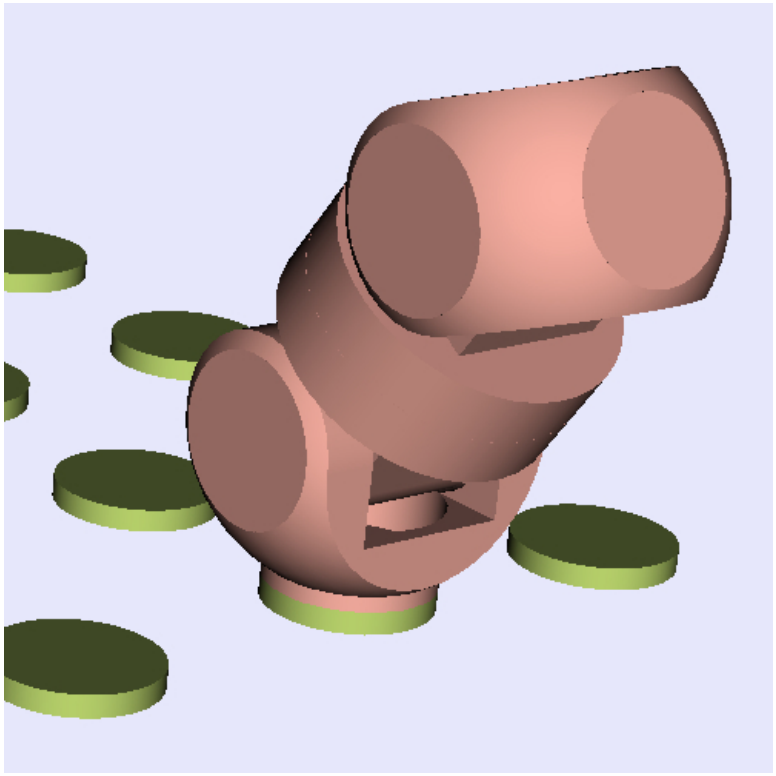

Figure 4.1: A universal module mounted on a pad (simulator).

Upon construction, the NetworkManager's instance initializes the networking; it creates network interfaces for each connector and an additional one for the virtual interface, and it sets up event listeners properly (i.e., listeners for events such as connector (dis)connect, arrival of a message via connector, etc.).

4.1.1 Managing interfaces

The Network Manager provides a set of functions for managing underlying components. For interfaces, it provides seven functions.

The first function is `findInterface` which takes a name of an interface and returns a pointer to an interface of a given name within the manager.

The second function is `interface` which takes the name as the previous function and returns a constant reference (i.e., an immutable

reference) to the interface of the given name. This function expects that the interface of a given name exists, otherwise an exception is thrown.

The third function is `interfaces` which returns an iterable range of all interfaces within the Network Manager. Ranges are a new concept introduced in the C++20 standard. It allows their users to abstract from concrete containers such as a vector or a list to an iterable collection of elements. The ranges provide additional functions that allow them to be easily composed and modified making them more suitable for interfaces like ours.

Additionally, there are functions `setUp` and `setDown` that both come in two versions. The first one expects an interface as an argument that should be set down or up respectively. The second version has no arguments and set all interfaces within the manager either up or down depending on the function called.

4.1.2 Managing protocols

Another component of the Network Manager are protocols. To list all of them available to the given instance of `NetworkManager`, there is a function `protocols` that returns a range of protocols that the Network Manager knows and can run on its interfaces.

To add a new protocol into the manager, we provide `addProtocol` function that accepts any protocol and adds it among the already known ones. The return type of the function is a pointer to the instance of the protocol running within the Network Manager. The pointer to such instance can be obtained by calling function `getProtocol` that takes a name of a protocol and returns a pointer to the running instance (or a `nullptr` if it does not).

To actually set protocols up on an interface, or to shut them down, there are functions `setProtocol` and `removeProtocol` respectively. Both functions expect the protocol and an interface on which the protocol should be set up or down.

4.1.3 Managing routing table

For the routing table as the last component we cover, the manager provides function `routingTable` that returns a constant reference to the routing table managed by the Network Manager.

Additionally, the Network Manager has functions `addRoute` and `rmRoute` that add or remove static routes from the table. Both functions expect an IP address of the network, its subnet mask, the gateway (outgoing interface), and the cost of such a route. The return type of these functions is a `bool`; the `true` value is returned if the route was properly added or removed, `false` is returned otherwise.

4.2 Interface

The network interfaces are implemented within the `Interface` class. The constructor expects a physical address (i.e., the MAC address) for the underlying network interface, an optional[1] connector and a callback function that is called on the connector's event (if present). If a connector is missing, the `Interface` creates a virtual interface.

The class also explicitly defines the destructor – a function that is called at the end of life of a given class instance. This is due to the resource management as the class manages the network interface for the underlying *lwIP* library, and has to be properly destroyed to prevent any memory leaks or incorrect resource access errors. Due to this 1:1 mapping, the `Interface` class does not allow making copies of itself. It is worth noting, that typically, user will not use these functions directly, instead the Network Manager will handle these itself.

There are other functions, which the user will use more regularly. There are basic functions for determining the state of the interface. Functions `isUp` and `isDown` query the interface state. There are functions `setUp` and `setDown` available for setting the state. Additionally, there is a function `isVirtual` that can be used to query the type of the interface (i.e., virtual or with an underlying connector), and a function `isConnected` that returns `true` value if the underlying connector is connected (for virtual interface it always returns `false`).

1. It is a `std::optional` that allows us to type-safely represent a value that may or may not be present. In case the reader is familiar with Haskell programming language, this is the C++ equivalent for `Maybe`.

One of the main goals of the new networking is the various protocol support. Protocols run on the individual interfaces, and for that reason, there are functions to set protocols up and down on a given interface. To set up a protocol on an interface, there is a function `setProtocol` that takes a listener address of a protocol (i.e., a multicast address) and a function, that is called on receiving a message destined for a corresponding protocol, and sets the listener appropriately. The oposite function is called `removeProtocol`, it takes a listener address and it removes the listener so the interface stops receiving protocol's messages. Moreover, there is a function `sendProtocol` thats takes a listener address and a packet, which then sends using a corresponding channel.

Furthermore, there are couple of functions to handle interface's addresses. Because the platform is using an IPv6 protocol by default, individual interfaces can have (and usually they do) multiple addresses at the same time. The maximal count of addresses depends on the underlying *lwIP* library, however, it can be configured via a macro[2] during the compilation process. The default count is three addresses per interface. The interface offers a function `getAddress` that accepts an index and returns a pair of an IPv6 address and its subnet mask. The function has also an argument-less variant that returns a vector of all addresses with their masks of a given interface. A new address can be assigned using a function `addAddress` that takes an address and a mask as arguments, and returns a `bool` value indicating if the addition was successful. The counterpart of this function is called `removeAddress` and its type is the same, only that it removes the input address.

Another way how an address can be assigned to an interface is to use a DHCP protocol. For IPv6, there exists DHCPv6 protocol which describes two possible ways of configuration; stateful and stateless [6]. The underlying lwIP library support the latter only. For this functionality, the Network Manager provides an interface via `dhcp` function, that can enable or disable dhcp on a corresponding interface. However, at the time of writing this thesis, the RoFI platform does not provide a DHCP server functionality.

2. The library is written in C language.

Finally, there is a function `send`, that expects a packet, and an 16-bit unsigned integer. Then, it sends given packet with a given content type[3] through the underlying connector. Eventually, all outgoing communication uses this function internally. In addition the interface traces the amount of data flowing through; using functions `dataSent`, and `dataReceived`, the user can determine how many bytes were carried over the appropritate interface. Due to the architecture of the underlying library, this functions are available for the non-virtual interfaces only (for the virtual interfaces, the functions return always zero).

4.3 Routing Table

The routing table is a core of the routing itself. In our implementation it is also the only component that defines additional classes. The main class is `RoutingTable` which has two other nested datatypes `Record` and `Gateway`. The `RoutingTable` holds a vector of `Record` internally, which symbolizes all known networks to the table. Each of those networks (i.e., each instance of a `Record` type) has, among other things, a list of `Gateways` that represent the individual paths to the given network. In addition to the list, each `Record` has an address and a mask – this determines a network. Each `Gateway` has an output interface and a cost which in combination with the corresponding `Record` unambiguously identifies a single route within the table. Moreover, the `Gateway` may include information about its origin – the protocol by which it was discovered; if such information is missing, the route was added manually.

The `Record` datatype not only keeps gateways, it also provides functions that simplify the work with individual routes. Functions `addGateway` and `removeGateway` can be used to add or remove new gateways with necessary checks for possible duplicates. There is a function `gateways` that returns an iterable range of all gateways within

3. The 16-bit number corresponds to a content type, which is an identifier for lower layers of networking. From the OSI/ISO model perspective, it is a field of the L2 layer header of the RoFI platform. However, that is not important, our networking works on a higher level (L3) and all of its messages are of the content type equal to 1. Other values may be used for messages such as firmware updates, however, in these cases our networking is not involved as it works on higher layers that do not have to be available for firmware updates and similar applications.

the record. The number of gateways can be determined via the function `size`. Additionally, function `best` returns a gateway wrapped in an optional type we have already seen – if the value exists, it equals to the gateway with the best cost for given network. At last, there is a function `compareNetworks` that accepts another record and decides, if the networks of both records match.

Moving to the actual table's interface, the main functions are `add`, `remove`, and `find`. The first two functions add or remove a given record respectively from the routing table. Their return type is a `bool` that indicates a change of the table after the requested operation is finished. The third function takes an address and a mask as its arguments, and returns a pointer to the corresponding record (if it does not exist, the pointer equals to `nullptr` as expected).

To make the work with table easier, the table provides additional functions that are usually used in case an interface becomes unavailable or a protocol is shut down on a certain interface. The first of these functions is `purge` that takes an interface's name and removes all gateways from all records within the table with this name. The second function is `removeInterface` that accepts a network (i.e., an address and a mask) and a name of an interface and removes all the gateways from the given network with the given interface as the output interface. Finally, the function `removeNetwork` expects an address and a mask as arguments and removes the record corresponding to given network. The last two functions return `bool` indicating the change of the routing table the same way as functions `add`, and `remove` above.

For protocols, there is a function `recordsLearnedFrom` that accepts a reference to a protocol and returns a vector of records that were added into the table via the given protocol. This function will be very useful when we will implement the concrete protocols in the next chapter.

4.4 Protocol

Until this moment, we described the individual implementations of Network Manager's components. For protocols, we are going to describe the implementation of the interface only and implementations for certain protocols will be presented in the next chapter.

The interface for protocols is represented by the class `Protocol`. The actual implementation of a protocol will be a separate class that will *inherit* from the `Protocol` class. The inheritance will ensure, that each derived class has to implement a certain set of functions as defined in the base class – `Protocol`. This class defines a set of functions to which it provides the default implementation, which means that the derived class (the implementation of a protocol) does not have to implement a given function by itself (however, it is possible to overrride the default implementation with its own, more efficient one, if needed). However, there are some functions, that are so-called *pure virtual functions*. This means that while implementing a derived class, the programmer has to provide an implementation for such a function. The fact that the `Protocol` contains pure virtual functions means, that the class itself is an *abstract class* and therefore cannot be instantiated, meaning that it is not possible to create an instance of such a class. This is a common approach when implementing interfaces in C++.

Once we have established the technique we use for the `Protocol` class, we move to the interface itself.

The first two functions that have to be implemented in each individual derived protocol are `name` and `address`. The first one returns a string representing the name of the protocol, the second function returns a multicast address on which the protocol accepts and to which it sends its update messages. Both values must differ among different protocols (e.g., the Network Manager's `addProtocol` functions checks this requirement before the new protocol is added).

On the contrary, there is a function `info` that returns a string containing basic information about the protocol – by default, the string contains the protocol's name and its address, but this behaviour can be overriden as we will see in the next chapter.

Protocols run on interfaces, therefore, the interface defines several functions that add or remove the protocol from an interface. The functions that provide this functionality are `addInterface`, and `removeInterface`; both take an interface as their argument and return `bool` indicating whether the request was successful. To determine if an interface is already running a protocol, there is a function `manages`.

We already mentioned that the state of the interface, including its addresses, may change in time. To inform a protocol about such a change, there are functions `addAddressOn`, and `rmAddressOn` accepting

an interface on which the change happened, an address added or removed and a mask as their arguments.

Once a protocol is running on an interface, it can send and receive messages via the interface. When a message for a protocol arrives, the function `onMessage` is called. The function accepts the interface on which the message arrived and the message (packet) itself. The return type of the function is `bool` that indicates if the protocol has any changes it needs to report.

We distinguish two types of updates; the first are routing updates (i.e., updates that potentially change the routing table), the second are configuration updates (i.e., updates that potentially change the state of interfaces). These updates can be accessed by functions `getRouteUpdates` for routing updates, `getConfigUpdates` for configuration updates respectively. There are functions to query if such updates exist; `hasRouteUpdates` and `hasConfigUpdates`, that does not take any argument and return a `bool` value according to the presence of corresponding updates. To clear all updates, there is a function `clearUpdates`.

After a message is processed and appropriate changes are made (by the `NetworkManager`, more on that in the next part of the chapter), then the protocol should be informed by calling of its function `afterMessage`. This message takes an interface and a function by which the protocol is able to send a packet through the given interface. Once again, the protocol can request changes by returning `true` from this function.

In addition to passing messages, the protocol has to be able to react on connection changes on interfaces it is running on. For that reason it provides a function `onInterfaceEvent` that takes an interface where an event took place and a flag indicating the nature of the event – either a new connection was established, or an old connection perished. As for the other functions, `true` is returned in case any changes for the routing table and/or iterfaces' configuration are required.

Finaly, there is a set of two functions that enable protocols to work with their records within the routing table. The first function is `setRoutingTableCB` that sets a callback function for the protocol, which it can use to obtain corresponding records (the routing table's `recordsLearnedFrom` function can be used), the second function is `routingTableCB` which invokes the callback.

4.5 The main logic

We have described interfaces of all components of the Network Manager including the interface of the class `NetworkManager` representing it.

Now we will focus on the inner logic of the `NetworkManager` class. We will not describe the whole solution as the source codes are provided along with this thesis, but we will focus on certain parts only.

In chapter 3, we introduced a scheme of handling protocol messages (figure 3.2). Now, we will describe the implementation of the scheme.

When a new protocol is introduced to an interface, the Network Manager's `setProtocol` function is called. The function then makes a few steps; 1) it sets a function that is called on message, 2) it adds the interface to the protocol, so that the protocol now manages the interface, 3) the Network Manager propagates updates of the protocol (if any).

The first step is to set up the listener function for the given protocol. For that, the Network Manager uses the interface's `setProtocol` function. The first argument is the protocol's listener address, the second one is the actual listener function – a function that is called upon receiving a message. This function receives several arguments, the only used is the packet (the message) itself. Because of the underlying *lwIP* library, when an interface listens on multiple addresses, the listener function is called for all protocols until one of them accepts the message. Due to this limitation, our listener function has to decide whether the packet is destined for the protocol it was set for, or not. This is realized by inspecting the destination address of the packet. If the address does not match the protocol, the function ends and the *lwIP* library calls a function listening on another address. If addresses match, then the function passes the packet to the protocol's `onMessage` function. After the protocol processes the message, the listener function checks its return value and if there are any changes, it will process them (e.g., if there are any route updates, then they are passed to the routing table properly). If the processing of those potential changes causes a change within the configuration of the networking, then the protocol is informed and its `afterMessage` function is called for all its managed interfaces. This is necessary as the routing protocol can get a

route update from one interface and then has to propagate this update via other interfaces to the rest of its neighbours. Possible additional changes are then handled in the same manner as previously.

The second step, after setting up the listener function on the interface, is to inform the protocol about a new interface to manage. This is realized by the protocol's function `addInterface`. The protocol will probably require some changes, so the Network Manager will process them and then it will clear the cache of the protocol by calling its `clearUpdates` function.

The third step consists of propagating the possible changes from the previous step. This propagation lies in calling `afterMessage` function of the protocol for all interfaces it manages.

4.6 Logging

As a part of the new networking, a provisional logging was implemented. The RoFI platform currently does not have a universal logging system, however, at least for debugging purposes a very simple logging was implemented. The main class is `Logger` and it supports adding logs with a few different severity levels.

All classes described in this chapter support this logging system, which is used by default. However, as stated previously, this is a temporary solution that is planned to be replaced by the platform-wise one.

We state its existence because it is a part of the source code provided along with this thesis, and it is mentioned in the final part of this chapter as well.

4.7 Command Line Interface

The last part of this chapter is dedicated to an optional extension of the implemented Network Manager. This extension is a command line interface (CLI) for the Network Manager that allows to control the Network Manager using text strings.

The interface is implemented in a separate header file, so if a user wants to use this functionality, a simple include enables it. The header file contains a single class `NetworkManagerCli` that has a single con-

tructor that accepts an instance of the `NetworkManager` class which it should manage.

The `NetworkManagerCli` class provides a simple interface with only a few functions. The first function is `command` that accepts a text string or an input stream (`std::istream`). The function then tries to interpret the input as a command and if it succeeds, then it executes the given command and returns `true`. Otherwise, an exception is thrown with a description of the error.

The second function provided is `help` that prints out a help message with a description of all supported commands.

All commands are of form `netmg <COMPONENT> command..` where the `<COMPONENT>` corresponds to one of a five components. These components are `interface` for control over interfaces, `table` for managing the routing table (including static routes), `forwarding-table` that enables to show the state of the underlying lwIP's forwarding table, `protocols` for protocol management, and `logs` for displaying logs of the networking.

This interface allows users to control the networking without any limitations and without a need of writting a CLI application for themselves. However, because this solution is implemented within a separate header file, the user still can implement the interface by himself, but is not forced to do so.

This interface is also used for the demo application we use for experiments and is described in the chapter 6.

5 Protocols

The previous chapter introduced the general interface for protocols. In this chapter, we will show how individual protocols can be implemented using the interface.

First, we present very simple Leader Elect protocol, that shows how the individual instances running on each module within the network can share information and based on their communication elect a leader. The leader then will obtain a second IP address on its virtual interface.

Second, we introduce implementations of two simple routing protocols – Simple Periodic and Simple Reactive – that differ in the way their update messages are shared. The first one shares messages on a periodic basis – hence the name – without unnecessary reactions to the changes in the network topology. The second one shares messages upon its initialization, but once the routes are shared, the protocol does not communicate until a network topology changes. The Network Manager's design allows both approaches.

Third, we show the implementation of the RoFI Routing Protocol – RRP, that was implemented within the bachelor's thesis as a part of the networking stack.

5.1 Leader Elect

We begin by specifying the functionality of the Leader Elect protocol. This simple protocol aims at electing a leader within the network of nodes running this protocol. Once the leader is elected, the protocol uses the interface provided by the Network Manager and assigns a leader's address to the virtual interface of a module elected as a leader.

The election process will be very simple; individual nodes will share their ID and the module with the lowest ID will be proclaimed the leader.

The Leader Elect protocol is implemented, as are all others protocols, in its own header file. The implementation consists of a class `LeaderElect` that inherits from the `Protocol` class as we described in the previous chapter. The constructor of the class accepts an ID of the

local module, and an address with a mask that should be assigned to the elected leader.

The protocol has to run on interfaces. Therefore, it implements necessary functions `addInterface`, and `removeInterface`, however, it does not offer implementations for functions `addAddressOn` and `rmAddressOn`, because the protocol does not work with (and therefore is not influenced by) the individual addresses and their changes. The protocol internally uses a database of interfaces that it manages (i.e., interfaces it runs on).

On interfaces that are added, and therefore managed by the protocol, the protocol exchanges messages. The format of Leader Elect's messages is very simple – a message consists of a single number that represents the lowest ID known to the corresponding module (and therefore the potential leader), that each instance of the protocol (one for every module) holds internally.

The processing of such messages is implemented in the function `onMessage`. The function starts by reading the incoming message containing the single number. Then, the number is compared to the internal number (the lowest known ID) that the receiving instance holds. If the numbers match, then the protocol marks the interface on which the message was accepted as converged, meaning that a neighbour behind the corresponding interface agrees on the potential leader, and the processing of the message ends. In the opposite case, when the potential leader IDs do not match, a few cases might occur.

The first case is that the neighbour's potential leader has a lower ID than the one currently held as a potential leader. In this case, the protocol updates its potential leader to reflect the lower ID and it proclaims all other interfaces as not converged.

The second case is that the current number is lower than the one received from the neighbour. In this case, it is necessary to inform the neighbour about a better (lower) ID in the network.

Additionally, when the incoming ID and the local ID match, and are equal to the ID of the local module, then the protocol requests a configuration change, and the leader's address gets assigned to it.

On the contrary, when the module has a leader's address assigned to it, and a newly arrived message contains an ID with a lower value, the protocol requests a change in the opposite direction, and the address is removed from its virtual interface.

The response messages are then sent by `afterMessage` function, that shares messages using only those interfaces, that are not marked as converged. This prevents an unnecessary communication between modules that are already in agreement about the lowest ID in the network.

To be formally correct, we should mention the listener address and the name of the protocol. The listener address of the Leader Elect protocol is defined as `ff02::aa:ff`, its name is `leader-elect`.

5.2 Simple Periodic

The next protocol we are going to present is a routing protocol. The first routing protocol, and the simplest one, is Simple Periodic protocol. This protocol shares routing updates with its neighbours, that consist of all the best records within the table, periodically with a little consideration of any updates of the network's topology. This has not only disadvantages in the unnecessary network congestion by updates, and most likely higher reaction times to topology changes that are not necessary (although this depends of the length of a period), but also it has certain advantages, mainly, periodic updates are more resistant to faulty connections. If a link over which two modules communicate is faulty (e.g., only a third of messages arrive to the second module), then the periodic updates will arrive successfully in a certain point, unlike protocols that share messages reactively unless they also implement some measures that deal with the possible unreliability, which add another level of complexity to the logic of such protocols.

The Simple Periodic protocol is implemented under a single class `SimplePeriodic`, that inherits its interface from the abstract `Protocol` class as well.

As the previous Leader Elect, the Simple Periodic protocol keeps track of all interfaces it manages as well. In contrast with the Leader Elect, it has to implement functions `addAddressOn` and `rmAddressOn` as well, because they affect the content of the routing table in case an interface is managed by the protocol. If a new address is added to an interface that is managed by the protocol, the newly added IP address has to be added into the routing table and shared with all neighbours. In case that an old address is removed, the corresponding

record in the routing table has to be removed as well, and all neighbours must be informed about the disappearance of the previously known route. These changes are handled via the interface defined by the `Protocol` class, namely these functions add a requested change among the changes that are then collected by the Network Manager after it calls aforemetioned function `getRouteUpdates`. The mechanism remains the same for the `addInterface` and `removeInterface` functions as well.

The main goal of a routing protocol is to share routes accross the network. The Simple Periodic protocol uses for sharing of those routes messages, that it sends through the individual interfaces it manages. When a message is received, `onMessage` function is called. The message contains a 16-bit number indicating how many routes are in the message, and the actual routes.

For each record, only the best route is sent. This has an interesting property in networks containing a loop. In those cases, for nodes on the loop, it may happen that not all nodes will have both routes (on a loop, modules are interconnected by two connectors – two gateways – and by each gateway all other nodes can be reached). This is due to the fact that we do not propagate routes we learned from an interface back to the same interface, so if the best route to a certain network leads through the neighbour to which we share an update message (and so by which we learned about the route), we skip the corresponding record even though there exists a second path through the opposite direction. This behaviour does not influence the reachability of modules within the network, even if the currently best route disappars because then the second best route will became the best and will be shared. It is worth noting, that the change in the protocol's source code required to change this behaviour is trivial in case the user would prefer the other approach.

Before the protocol starts adding routes it recevied, it has to remove all routes if obtained previously from the routing table via the interface on which the message was recevied. The protocol exchanges the whole table, so the routes that remain valid will be added in the next step, those that are no longer valid will stay removed. The routes the protocol added into the table can be obtained from calling the `routingTableCB` function that the Network Manager sets to call the

Routing Table's `recordsLearnedFrom` function with appropriate arguments. Hence, the protocol knows which routes to remove.

Processing the received routes is straightforward as the message contains their count and we know the structure of the individual route – an address, a mask, and its cost. Before adding the route into the table, the protocol can modify the cost (however, it can be modified upon sending, both approaches are possible). For Simple Periodic, it was arbitrarily decided that for each hop the cost will increase by 10.

After processing all routes, the Network Manager withdraws all updates from the protocol using the `getRouteUpdates` function and passes them to the routing table. Once the updates are applied, the Network Manager calls Simple Periodic's `afterMessage` function implementation.

In case `afterMessage` was called previously, nothing happens and no message is sent as updates are shared periodically, not in response to a recevied message. In case the function is called for the first time, periodic updates are set up. As its arguments, the function takes an interface and a function that can be called to send a packet using the given interface. The Simple Periodic protocol stores the function internally, and notes the `afterMessage` function was already called, then, it sets the periodic updates.

Periodic updates are realized by a single function that at first sends a packet with the current content of the routing table, at second, it sets a timer after which the function is called again. The timer is set to the period of the Simple Periodic instance that is determined upon its construction, and for the timer itself is used RoFI platform's `wait` function, therefore, this solution is versatile and works on physical modules, but also in platform's simulators.

As for the previous protocol, we note that the Simple Periodic's listener address is `ff02:a:b:b:af`, and its name is `simple-periodic`.

5.3 Simple Reactive

The second simple routing protocol is the Simple Reactive protocol. It is implemented in class `SimpleReactive` and it works very similarly to the Simple Periodic protocol, but its updates messages are shared differently – after the initial message(s) exchange, the protocol remains

silent until the network's topology changes. This may have a positive effect on the network congestion, but also can play a role in the power consumption of the RoFIbot.

The main parts of the Simple Reactive protocol are implemented the same way as their counterparts of the Simple Periodic protocol, therefore, we will restrict our description only to those parts that differ.

The first main difference can be found in the implementation of `afterMessage` function. In the previous protocol's implementation, the periodic timer was set up. In the Simple Reactive protocol, the function only sends a given message by passing the message – its first argument – to the send function – its second argument.

The second difference is an approach we have to chose in `onMessage` function. Previously, we initially removed all records from the routing table for the appropriate interface, and then we added all routes we received in the message. We could use this simplier approach because even though the `afterMessage` is called always the routing table is changes, its implementation did not actually send any messages. The implementation of the `afterMessage` no longer allows this approach, so each time a message is received, we process all routes within the message and we add only those, who are not yet in the table. Also, we remove all routes that are in the table, but was not included in the update message as this means that they are no longer valid.

The second difference lies in the necessity of the protocol to react on potential connection changes (i.e., situations, when the underlying connector of a managed interface is newly connected or disconnected). In case a new connection appears, the protocol has to send an update message via this new connection and process the potential response (and then, if any routes are added, inform the rest of its neighbouring modules). In case a connection perishes, the protocol has to remove all routes using this interface as an output interface, because the paths are no longer available. Additionally, it has to inform the neighbouring modules as well, so that they will not hold non-existing routes in their tables.

The rest of the protocol's implementation mirrors the implementation of the `SimplePeriodic` class. Once again, we specify that the Simple Reactive's listener address is `ff02:a:b:b:ae`, and its name is `simple-reactive`.

5.4 RRP – RoFI Routing Protocol

The last routing protocol whose implementation we will provide is the RoFI Routing Protocol introduced simultaneously with the former networking.

In comparison with the previously described protocol, the RRP has a few advantages. The first advantage is that the updates do not contain the whole table (except the initial messages), but they contain only changes between the current and the previous message. At least in theory, this can reduce the size of those updates. The second advantage are so-called stub areas – areas, that have only one path towards the rest of the network. This fact can be then used to realize that to the stub area it is not necessarily needed to send updates about the rest of the network as the path will always lead one way. Such a feature might have a positive influence on the number of update messages flowing through the network.

At the same time, it is worth noting that these features have certain implications such as increased demands on resource (mainly on memory) of the individual modules as the protocol has to hold more information in each instance.

The implementation of RRP is in a class of the same name – `RRP`. Because RRP itself shares its update messages reactively in the same way as Simple Reactive protocol, it shares the implementation of certain functions with it. Among those functions are primarily those which handle interfaces, such as `addAddressOn`, `rmAddressOn`, `addInterface`, and `removeInterface`.

The main difference lies, therefore, in functions `afterMessage`, and `onMessage`. Before we start with describing of these messages, we will describe the inner state of the protocol – everything the protocol keeps internally in addition to Simple Reactive protocol.

We already established that RRP shares only changes that happen since the last change. For this purpose, the protocol keeps two pools for two types of updates. The first one if interface related and it contains those updates, that should be shared via this interface only (e.g., a response to an update message). The second one works in the opposite direction, it contains updates that should be shared by all but one interface (e.g., a new route to be share with all neighbours except the one from which it originated). Moreover, the protocol keeps a

command for each interface, corresponding to the command that should be included in the next message.

These commands are part of a header of the update messages, so that RRP has a wider range of possible reactions to an incoming message. There are six types of commands, thus six types of the update messages. The first pair are *Hello* and *Hello Response* messages that contain the best records within the routing table. The second pair are *Call* and *Response*, when the latter is the response to the former. Both contain only the differences between subsequent *Calls*. The remaining two types are *Stubby* and *Sync* that are tied to the stub area feature.

Stubby areas

The stub areas are those areas, that have only one gateway towards a non-stubby part of a network. In other words, module is *stubby* (part of a stub area) if all records in his routing table have only one gateway (except the virtual one), or all except one are pointing to a stubby area (the virtual interface is not considered).

Whereas update messages in non-stubby parts of the network propagate very similarly as in the case of Simple Reactive protocol, in RRP's stubby areas the policy is different. When a module discovers it is in a stubby area, it sends a message with a *Stubby* command to the only (or the only non-stubby) gateway it knows. The neighbour then marks the corresponding interface as a stubby interface (i.e., leading to a stubby area); for this purpose the protocol keeps a flag for every interface it manages. If the module realizes it is a stubby module whlist responding to a message, the proper command is *Sync*. This module will no longer receive any update messages from the rest of the network (with an exception of the topology changes, that lead to violation of the stub condition, and therefore to an end of the stubby area).

However, to secure that the stubby areas are able to reach the rest of the network without any updates, the module's routing table must be modified. In case the module has no other neighbours than the gateway, it will only remove all routes and add a single one – a default gateway represented by an address and a mask `::/0`. In case the module has some other stubby neighbours, it will keep route to those modules within the table, remove the rest and add the default

gateway as well. The default gateway is a route that server as a last option, therefore, if there is any other route to a destination, it will be preferred over the default gateway (the routing process preferes addresses with higher masks). This ensures that the packet will always reach the network, where the routing tables contain more information.

Update messages

To correctly populate routing tables of modules in the rest of the network, the update messages are sent. As we already mentioned, they contain the command signalizing the type of the message. They also contain the number of record's updates in the message. However, the individual records slightly differ over the previous protocols. Because RRP shares only changes, each record has to have a flag about the change it represents. We difference only two types of operation; `Add` and `Remove`. The rest of the single record remains the same; it has an address, a mask, and a cost.

The messages are sent, as with other protocols, in `afterMessage` function. The function manages a few responsibilities. First, when an update processed in `onMessage` causes the module to become stubby, the `afterMessage` function sets appropriate flags to all interface as described above and handles the necessary changes in the routing table. Second, when the function is called on an interface pointing to a non-stubby interface, and there is a command prepared for the next message on a given interface, then an update message of the given type is sent. The message itself is practically the same in case of *Hello* and *Hello Response* messages as for the previous protocol; for the other types of messages, the only difference lies in the fact that instead of the routing table, the source for the records are the two internal update pools. Third, the function also erases the command flag on the given interface to indicate the message was already shared (and no other will be permited until the flag is set anew).

The last function to discuss is `onMessage` function. This function has to not only process the individual update messages, but also set correctly the commands for individual interface and their response messages. At start, the process works as before. The protocol reads the message type and the count of records in the message. Then individual records are read and added among the updates for the routing table.

Then, based on the current state of the module (is it a stubby module?), and the type of the message that arrived, the type of the response is determined. Also, if an update affecting the rest of the network arrived, the rest of the interfaces' commands has to be determined as well.

Finally, we note that RRP uses `ff02:1f` address as did its implementation in the bachelor's thesis, and its name is `rrp`.

5.5 Summary

In this chapter, we introduced the individual protocols implemented within the Network Manager. To honor the modular architecture of the solution, each protocol is implemented within its header file but no protocol is by default enabled or even part of a newly created instance of the `NetworkManager` class as it is the user who should add protocols he wishes to use.

In the next chapter, we will use the implementations of protocols described in this chapter to show how they work in a set of demo applications available on the platform.

6 Experimental evaluation

This chapter is dedicated to the experiments and demo application of the presented Network Manager, and is divided into several parts.

In the first part of this chapter, we present two demo applications using the `NetworkManager` class to outline the possibilities the solution provides. The first application will be a re-write of an application already present in the original networking to show how simple it is to move to the new networking. The second application is an application that is tailored to the Network Manager itself and aims to show some of its potential, including its optional command line interface.

In the second part of this chapter, the individual protocols implemented in the previous chapter will be experimentally checked on multiple different network topologies.

6.1 Demo applications

6.1.1 Communication example

The platform contains in its codebase several example applications to not only show some of its functionality but also provide a template for writing applications for the platform using certain features.

The communication example is a straightforward way to present how modules can be set up and exchange messages. The application itself runs per module (i.e., a binary executing the application is uploaded into the module, where it is then executed), or in the case it is run within a virtual environment (i.e., one of the simulators), one instance is started for each individual module within the environment.

At the start, the application assigns an IP address to the virtual interface of a corresponding module and then decides whether a simple server will be run (for a module with an ID equal to 1) or the module will start a client (for the rest of the modules). The server starts listening for incoming UDP[1] connections on a previously agreed port. When a request is received, the server prints the address of the sender and a message the received packet contains. Then it sends a

1. Both TCP and UDP are supported.

message back to the sender. The client-side consists of a single loop that periodically sends messages to the server.

In order to use the Network Manager, only a few lines of code must be changed. The former networking initialization is the following:

```
auto localModule = rofi::hal::RoFI::getLocalRoFI();
rofinet::RoIF roif( localModule );
roif.setUp();
```

The `RoIF` class initializes the individual interfaces and assigns an address based on the local module's ID. It also sets the RRP up. The method `setUp` then sets all interfaces up and therefore allows data to flow.

The same effect can be achieved using the Network Manager as well with the following code:

```
// assume using namespace rofi::net;
auto localModule = rofi::hal::RoFI::getLocalRoFI();
NetworkManager net( localModule );
auto* proto = net.addProtocol( RRP() );

if ( id == 1 ) {
    net.addAddress( "fc07:0:0:1::1"_ip, 64
                  , net.interface( "rl0" ) );
} else if ( id == 2 ) {
    net.addAddress( "fc07:0:0:2::2"_ip, 64
                  , net.interface( "rl0" ) );
} else if ( id == 3 ) {
    // ...
}

net.setUp();
net.setProtocol( *proto );
```

The code is obviously larger; however, most of it consists of conditional expressions that assign corresponding addresses to the module's interface. This was originally part of the networking stack initialization but was removed from the Network Manager. The rest of the code grew only by a small amount. The first two lines are almost identical. They

differ only in the class they use to provide the networking. The second code snippet contains a line calling `addProtocol` function to add `RRP` instance into the Network Manager. Then, after the addresses of modules are set up, interfaces are brought up using the `setUp` function analogously to the previous solution. Finally, `setProtocol` function must be called to start the process of routing using the RoFI Routing Protocol.

Although the solution using the Network Manager takes slightly longer to achieve similar functionality, its advantage lies in the functionality that the former solution does not have. For example, changing the used routing protocol is impossible in the case of the former solution. With the Network Manager, only a single line is needed to change

```
// assume using namespace rofi::net;
auto localModule = rofi::hal::RoFI::getLocalRoFI();
rofi::net::NetworkManager net( localModule );
auto* proto = net.addProtocol( SimpleReactive() );

// < address assignemt ommited >

net.setUp();
net.setProtocol( *proto );
```

to use a different routing protocol (in this case, the Simple Reactive) while the rest of the application's code base remains the same.

Furthermore, to show how multiple protocols can be run at the same time, we modified the communication example also to use the Leader Elect protocol. Before, the address of the server was the address of one of the modules. In the new version, the server's address will be the address that is assigned to the leader elected by the Leader Elect protocol. Besides changing the address of the server from the module's address to the one the protocol sets, it suffices to add a few lines only:

```
// assume using namespace rofi::net;
auto localModule = rofi::hal::RoFI::getLocalRoFI();
NetworkManager net( localModule );
```

```
auto* proto = net.addProtocol( SimpleReactive() );
LeaderElect leaderElect( localModule.id()
                       , addressToAssign
                       , maskToAssign );
auto* proto2 = net.addProtocol( leaderElect );

// < address assignemt ommited >

net.setUp();
net.setProtocol( *proto );
net.setProtocol( *proto2 );
```

to achieve the desired functionality. The Leader Elect protocol decides based on the ID, so the module with the lowest ID will be the server as before; however, in the case we would like to choose the leader differently, it is enough to implement another protocol for a leader election and then change the single line to use it instead of the currently used Leader Elect. The whole code can be found in the source code attached to this thesis (more details are in the last chapter).

6.1.2 The Network Manager CLI application

The second demo application we show is an application demonstrating the potential of the Network Manager itself and can be used to test and explore all features that are provided.

The application starts with the initialization of the networking for the corresponding module. Consequently, it assigns an IP address of `fc07:0:0:ID::1/64` to each module, and it adds all protocols whose implementations we have discussed in the previous chapter to the Network Manager. It also sets up a listener listening for any UDP connections on a given port.

Furthermore, an interactive prompt is presented to the user. There, the user can type commands that it wants to be executed. Three types of commands are supported.

The first type is commands designated to the Network Manager itself – its CLI. These commands start with `netmg` and have the form outlined at the end of chapter 4. Using this interface, the user can run individual protocols on any given interface, add or remove their

addresses, or even add or remove static routes within the routing table.

The second type is represented by commands `msg` or `message` that allow the user to send any message to any address. If a route to the supplied address is known, the message is sent and displayed in the output of an instance running on the corresponding module. If no route exists, then the user is informed about such an issue.

The last type is commands `connect` and `disconnect`, which allow the user to connect or disconnect individual connectors of the corresponding module. This proves to be useful if a user wants to check that the routing table is updated according to the network changes.

This application is also used in the next section of the chapter, where we focus on protocols and their behaviour in different network topologies.

6.2 Experiments – protocols

We have described multiple protocols within the previous chapter. In the following part of this chapter, we will experimentally validate their behaviour in multiple different networks. First, we describe the set of networks we will use in our experiments, then we will validate the Leader Elect protocol, and finally, we will focus on all routing protocols. Everything will be realized through *simplesim* – a platform's built-in physics-less simulator. Alternatively, a physical simulator based on Gazebo simulator developed by Ondřej Svoboda [7] can be used; however, for networking purposes simulating physics does not bring any additional advantages.

6.2.1 Various networks

To simulate various topologies networks can form (i.e., set of modules), we have designed four different sets of modules which we call subsequently; a) snake, b) circle, c) lasso, d) tree. These sets are displayed on figures below and numbers in those figures correspond to IDs of modules next to them.

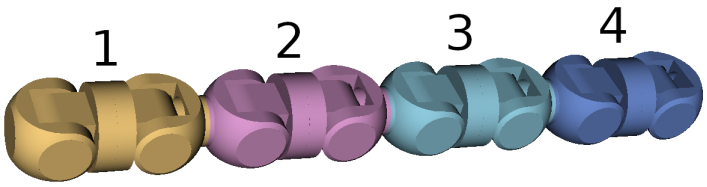

Figure 6.1: Test network I – snake.

The snake topology (shown in figure 6.1) contains four modules that are connected sequentially to each other. This represents the simplest network (if we ommit the trivial one with a single module).

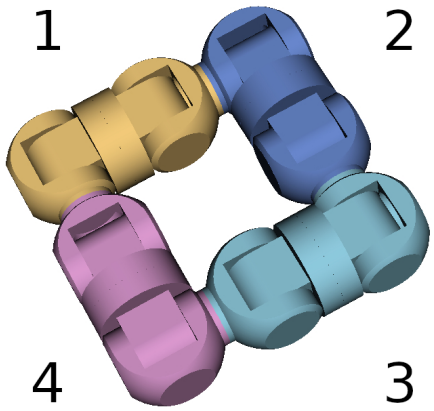

Figure 6.2: Test network II – circle.

The circle network (figure 6.2) consists of four universal modules. Those modules are connected to form a simple circle. This means that to each other module, there exist two paths to the remaining three modules. It also represents more complicated topology for routing protocols.

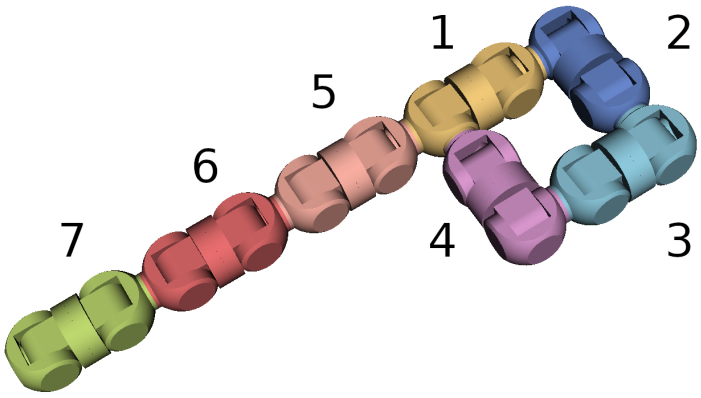

Figure 6.3: Test network III – lasso.

The lasso topology 6.3 combines both, snake and circle topologies containing sevem universal modules. In this topology we expect to see differences between the simplier protocols, Simple Periodic and Simple Reactive, and RRP, that supports stub areas. We expect stub area to form itself in the straight part of the network.

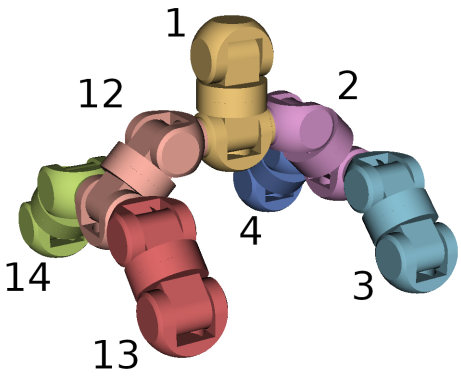

Figure 6.4: Test network IV – tree.

The last topology is in a shape of a tree, and is shown in figure 6.4. It does not contain any loops, but its seven modules use more connectors to communicate as paths within the network branch wide.

6.2.2 Leader Elect

In case of the Leader Elect protocol, the experimental validation will not be very long. At first, we run the protocol on all interfaces and all modules, then we check if the leader was elected, and whether the appropriate address – `fc07:a::a/96` was set to the right interface.

The initial state and configuration of the module 1 before the protocol was set up[2]:

```
Starting NetworkManager example
ID: 1
address: FC07::1:0:0:0:1
> netmg interface show
rl0 [up] , rd1 [up] , rd2 [up] , rd3 [up] , rd4 [up]
    , rd5 [up] , rd6 [up]
> netmg interface rl0 show
interface rl0
         address 0: FE80::101:101:101 [link-local]
         address 1: FC07::1:0:0:0:1/80

> netmg interface rd2 show
interface rd2
sent: 0 bytes / received: 0 bytes
         address 0: FE80::301:1FF:FE01:101 [link-local]
```

We see that all interfaces are up, and virtual interface has the global address. The virtual and the rest of interface have the link-local addresses as required for IPv6 devices.

After the protocol was set up on all interfaces and all modules within the network, the configuration of the virtual interface changed appropriately.

```
> netmg interface rl0 show
interface rl0
         address 0: FE80::101:101:101 [link-local]
         address 1: FC07::1:0:0:0:1/80
         address 2: FC07:A::A/96
```

2. The format of the output is intentionally kept in the same way as the user will encounter using the demo application.

As we can see above, the address was assigned to the virtual interface with index 2. The rest of modules within the network remained intact. As we can see on the output from the module 4, the only change is that the local instance of the protocol correctly notes the leader's ID.

```
> netmg protocol leader-elect show
leader-elect; address: FF02::AA:FF; leader id: 1 leader
    address: FC07:A::A
> netmg interface rl0 show
interface rl0
         address 0: FE80::404:404:404 [link-local]
         address 1: FC07::4:0:0:0:1/80
```

The same holds for all other network topologies as well. In case the Leader Elect protocol is enabled first at modules with a higher IDs and then a module with a lower ID is introduced to the network (or more precisely to the election process by starting the protocol), the leader changes, and therefore the address from the previously elected leader is removed correctly and added to the new one.

6.2.3 Routing protocols

Now we will move to the routing protocols. This part of a chapter is divided into four sections reflecting the number of the test networks. Alternatively, we could divide this part into section by individual routing protocols, but this way the differences between protocols stand out better.

The process of running those experiments is same for all networks and all protocols. At first, all instances for individual modules are set up. Then, starting from a module with the lowest ID (1) to the module with the highest, the chosen protocols are started on all interfaces of the given module. After that, routing tables are explored. Additionally, actions such as disconnecting/re-connecting of connectors may take place with further routing tables exploration.

Network I: Snake

The first network is the snake whose layout is captured in figure 6.1. The first protocol we are going to discuss is the Simple Periodic proto-

col with period of 4 seconds. Also, each hop between two modules adds 10 to the path's cost.

After setting everything up, the routing tables of all modules contain records about all other modules. For example the routing table of module 2 has the following content:

```
> netmg table show
FC07::2:0:0:0:1/80:
                          via rl0 : 0
                              rl0 : 0  [from simple-periodic]
FC07::3:0:0:0:1/80:
                          via rd3 : 10 [from simple-periodic]
FC07::4:0:0:0:1/80:
                          via rd3 : 20 [from simple-periodic]
FC07::1:0:0:0:1/80:
                          via rd6 : 10 [from simple-periodic]
```

When the connection between modules 2 and 3 is disconnected, the routing tables are updated appropritately (i.e., modules 1 and 2 loose routes to modules 3 and 4, and vice versa). After re-connecting the two modules, it takes up to 4 seconds (the length of the period) until all routing tables contain routes to all other nodes within the network.

This is the key difference between the Simple Periodic and two remaining protocols, that react to the changes immediately. Otherwise, the routing table of Simple Periodic and Simple Reactive protocols do not differ (except costs of routes, as Simple Periodic adds only 5 points to a cost for a hop).

Considering routing tables, there is also a major difference between RRP and the other two protocols. When we compare the routing tables of modules at both ends of the network, we will notice an effect of the stub area support.

```
# Module 1
> netmg table show
FC07::1:0:0:0:1/80:
                          via rl0 : 0
                              rl0 : 0 [from rrp]
```

```
::/0:
                        via rd3 : 0 [from rrp]

# Module 4
> netmg table show
FC07::4:0:0:0:1/80:
                        via rl0 : 0
                            rl0 : 0 [from rrp]
FC07::3:0:0:0:1/80:
                        via rd6 : 1 [from rrp]
FC07::2:0:0:0:1/80:
                        via rd6 : 2 [from rrp]
FC07::1:0:0:0:1/80:
                        via rd6 : 3 [from rrp]
```

As we can see, the first module contains only the default gateway as it is at start of the stub area. With each following module, the routing table must contain, besides the default gateway, also the routes to the other end of the stub area as we can see on the routing table of module 3.

```
# Module 3
> netmg table show
FC07::3:0:0:0:1/80:
                        via rl0 : 0
                            rl0 : 0        [from rrp]
FC07::2:0:0:0:1/80:
                        via rd6 : 1        [from rrp]
FC07::1:0:0:0:1/80:
                        via rd6 : 2        [from rrp]
::/0:
                        via rd3 : 0        [from rrp]
```

In case the routing table did not include those routes, the communication within the stub area would not be possible.

Network II: Circle

The next network is in the shape of a circle. This kind of network is more difficult to handle because updates not only arrive from both directions, but they also contain the same networks.

The first protocol we examine is the Simple Periodic protocol. As previously, the Simple Periodic protocol successfully shares the routes to individual modules accross the network. It might be note worthy that in the circular topology, we can observe consequences of our decision to share only the best routes. For example, the routing table of module 3 does not have the second route to the module 2.

```
# Module 3
> netmg table show
FC07::3:0:0:0:1/80:
                        via rl0 : 0
                            rl0 : 0  [from simple-periodic]
FC07::1:0:0:0:1/80:
                        via rd2 : 20 [from simple-periodic]
                            rd6 : 20 [from simple-periodic]
FC07::4:0:0:0:1/80:
                        via rd2 : 10 [from simple-periodic]
                            rd6 : 30 [from simple-periodic]
FC07::2:0:0:0:1/80:
                        via rd6 : 10 [from simple-periodic]
```

This is due to our decision to share only the best route with the condition of not sending the routes by the same interface they arrived at (in that case, an infinite loop of messages would ensue). This does not have any effect on the actual availability of individual nodes, the missing route has triple the cost and will be shared in the case the current route disappears.

Moving to Simple Reactive protocol, the routing tables look alike. The same behaviour towards the availability of the worse routes as for Simple Periodic protocol manifests. The Simple Reactive protocol maintains its ability to converge faster as it is not limited by the number of messages a module can send by a given period.

For RRP, the process becomes more challenging. Stub areas are not well suited for such circular networks, so that we can get results such as is the routing table of module 3.

```
# Module 3
> netmg table show
FC07::3:0:0:0:1/80:
                          via rl0 : 0
                              rl0 : 0 [from rrp]
::/0:
                          via rd6 : 0 [from rrp]
FC07::4:0:0:0:1/80:
                          via rd2 : 1 [from rrp]
```

In a certain point the protocol decided that it might create a stub area and therefore replace the rest of the records by a single default gateway. Even though the routing table is less readable than in the previous cases, it still contains all information necessary to communicate with any node within the network.

Network III: Lasso

The next network combines features of the previous two into a single one. It also consists of more modules; instead of four, there are seven.

As usual, we start with the Simple Periodic protocol. It populates all tables as expected, however, even though the network is not big by any means, the period of 4 seconds results in noticeable delay. This can be of course mitigated by lowering the period, however, by its nature this approach will be always as fast as the reactive updates at best. The Simple Reactive protocol manages to populate all routing tables as well.

The RRP gives us mixed results. On one hand, the routing tables are again less user friendly than tables of the other two protocols. However, the protocol proclaims the straight part of the lasso to be a stub area, so for example the module 7 can have only one external path represented by the default gateway and yet be able to reach every other module in the network.

Network IV: Tree

The tree structure of the last network is once again a loop-free consisting of seven modules.

Beginning with Simple Periodic protocol, we obtain routing tables populated as expected. The routing table of the module 1 highlights the symetry of the network.

```
# Module 1
> netmg table show
FC07::1:0:0:0:1/80:
                        via rl0 : 0
                            rl0 : 0  [from simple-periodic]
FC07::12:0:0:0:1/80:
                        via rd2 : 10 [from simple-periodic]
FC07::13:0:0:0:1/80:
                        via rd2 : 20 [from simple-periodic]
FC07::14:0:0:0:1/80:
                        via rd2 : 20 [from simple-periodic]
FC07::2:0:0:0:1/80:
                        via rd1 : 10 [from simple-periodic]
FC07::3:0:0:0:1/80:
                        via rd1 : 20 [from simple-periodic]
FC07::4:0:0:0:1/80:
                        via rd1 : 20 [from simple-periodic]
```

Subsequently, using Simple Reactive protocol yields the indentical results except the lower costs.

On the contrary, the RRP protocol does not populate the routing tables correctly. There exist some modules that are missing from the routing tables (e.g., module 14 from the module 1's table). We suspect it is due to incorrect implementation of stubby areas, however, at the time of writting this thesis, the issue was not solved.

6.3 Discussion and Limitations

We have experimentally verified the implemented solution. The Leader Elect protocol behaves as expected in all four networks we used.

For the routing protocols, the behaviour varied mainly with the RoFI Routing Protocol. The main issue are stub areas that it supports and tries to incorporate to the routing process. However, as we have seen, in certain cases the routing tables are harder to read and the final state of the tables is more susceptible to external conditions (e.g., the time interval between start of individual instances, etc.). Simple Periodic protocol achieved more consistent results, yet by its nature more slowly than the Simply Reactive protocol. The last mentioned protocol provided consistently the best results overall. Its routing tables were easier to read than those of RRP, but at the same time it provided results faster than the Simple Periodic.

With the above in mind, the main purpose of this chapter was not to thoroughly evaluate the individual protocols, but to show capabilities of the Network Manager. We have shown how to use the interface to write our own routing protocols. We have also shown two demo applications, one where we used the Network Manager instead of the original networking, the second to show that the new networking migh be used even interactively without sacrificing the functionality provided.

The main limitations of this chapter are not only a small sample of networks we used, but the fact we restricted ourselves to the simulated environment. At the time of writting this thesis, we do not have enough of physical modules to verify those experimets in real world.

Another limitation worth mentioning is the lack of higher level applications and algorithms. We verified that it is possible to share routes across the network to communicate, however, this kind of communication is usually the basis which more sophisticated applications and algorithms are built upon.

7 Conclusion

We introduced and implemented the Network Manager tool into the RoFI platform. The Network Manager allows better control over the network stack of the platform bringing manageable networking to its users. With respect to the former networking solution, the Network Manager is more extendable and provides more functionality. Its components are also better distinguished, so its future development might be easier.

We also described the general interface for protocols. Furthermore, several protocols were implemented to illustrate Network Manager's capabilities and also to offer a blueprint to potential users about how the implemented solution can be used.

In later chapters, we introduced two demo applications where one was the evolution of a previously implemented application using transport layer protocol, and the second was a command line application capable of network management. Using the latter one, we then applied the implemented protocols to several different networks.

The full implementation of the work is provided along with this thesis. The source code is also available on GitHub in the platform's repository `https://github.com/paradise-fi/RoFI` on the *network-manager* branch. The implementation of the Network Manager itself is located in `softwareComponents/networking/` including the individual protocols. Sets of modules used in chapter 6 can be found under `data/configurations/json/` and the demo applications located under `examples/simulator/`. The implementation is also documented. The documentation can be built along the source code or can be viewed online `https://paradise-fi.github.io/RoFI/branch/network-manager/rofilib/`.

7.1 Future work

As we already discussed in the previous chapter, there are still things to improve. We have seen that the stub areas of RRP in its current implementation do not achieve the desired goals.

Another area whose development could be beneficial is applications based on networking. The platform is currently missing NTP

or DHCP servers. We have also discussed the missing platform-wide solution for logging.

The presented solution was tested in the simulated environment only. Even though this environment should reflect the behaviour of the physical modules, it would certainly be wholesome to test the same scenarios once the hardware is available.

Bibliography

1. MRÁZEK, J. *RoFI – Distributed Metamorphic Robots* [online]. Brno, 2018 [visited on 2022-12-06]. Available from: `https://is.muni.cz/th/y1s7e/`. Faculty of Informatics, Masaryk University. Supervised by Jiří BARNAT.
2. CHLUP, V. *Routing within RoFI Platform* [online]. Brno, 2020 [visited on 2022-12-06]. Available from: `https://is.muni.cz/th/h6jtl/`. Faculty of Informatics, Masaryk University. Supervised by Jiří BARNAT.
3. MRÁZEK, Jan. *A universal module.* 2022. [Personal archive].
4. *lwIP library* [online] [visited on 2022-12-07]. Available from: `https://savannah.nongnu.org/projects/lwip/`.
5. *NetworkManager, official source code repository* [online] [visited on 2022-12-07]. Available from: `https://gitlab.freedesktop.org/NetworkManager/NetworkManager`.
6. T. MRUGALSKI M. Siodelski, ISC B. Volz A. Yourtchenko Cisco M. Richardson SSW S. Jiang Huawei T. Lemon Nibbhaya Consulting T. Winters. *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)* [Internet Requests for Comments]. 2018 [visited on 2022-12-11]. Available from: `https://www.rfc-editor.org/rfc/rfc8415`. RFC.
7. SVOBODA, Ondřej. *Simulating RoFI Platform in GazeboSim*. Brno, 2020. Available also from: `https://is.muni.cz/th/y1rvd/`.