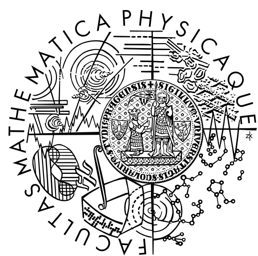


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Stanislav Gálfy

HelenOS routing and porting of BIRD

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Mgr. Martin Děcký, Ph.D.
Study programme:	Informatics
Study branch:	Software Systems

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague on 10.05.2018 signature of the author

Title: HelenOS routing and porting of BIRD

Author: Stanislav Gálfy

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Ph.D., Department of Distributed and Dependable Systems

Abstract: Capability to route can be considered as one of the key features of a modern multipurpose operating system, which HelenOS aims to be. The goal of this master thesis is to explore current HelenOS routing capabilities, enhance them and empower HelenOS with BIRD. HelenOS will become a routing operating system with awareness of its surroundings. It will be capable of dynamic adaptation to changes in the network and their propagation.

Keywords: HelenOS, BIRD, networking, routing, microkernel

I would like to thank my supervisor, Martin Děcký, for his guidance throughout this research. I would like to thank HelenOS developers Jakub Jermář, Jiří Svoboda and Vojtech Horký for their work on HelenOS and the time spent on articles about it. Also, I would like to thank CZ.NIC developers for creating BIRD.

Contents

Introduction **3**

1 BIRD **4**
1.1 Version . . . 4
1.2 BIRD modules . . . 4
1.3 Execution flow . . . 5
1.4 Threads . . . 6
1.5 System dependent parts . . . 6
1.6 Protocols . . . 9

2 HelenOS **11**
2.1 Network stack . . . 11
2.2 Transport layer . . . 12
2.3 Network layer . . . 13
2.4 Link layer . . . 15
2.5 Coastline . . . 16

3 Analysis **17**
3.1 Testing environment . . . 17
3.2 Porting BIRD . . . 18
3.3 HelenOS socket design . . . 22
3.4 Sockets requirements . . . 24
3.5 Additional requirements . . . 25
3.6 BIRD's HelenOS system dependent layer requirements . . . 27

4 Implementation **28**
4.1 Debugging techniques . . . 28
4.2 Socket implementation prequisities . . . 28
4.3 Socket POSIX library . . . 28
4.4 Socket C library . . . 29
4.5 Socket server upper layer . . . 30
4.6 Socket server lower layer . . . 31
4.7 Additional changes . . . 36
4.8 BIRD's HelenOS system-dependent layer . . . 39

5 Evaluation **40**
5.1 Environment setup . . . 40
5.2 Tests . . . 41

Conclusion **46**

Bibliography **47**

List of Abbreviations **48**

Appendices **49**

A Electronic attachment **50**

B Compiling and running **51**

C Detailed configuration of autonomous systems **53**

Introduction

Motivation and goals

This thesis aims to describe the upgrade of HelenOS from an operating system that can act only as a simple network endpoint to a routing operating system with some advanced features.

HelenOS is a modern multipurpose operating system mainly used as a platform for academic research. Upgrading it to a routing operating system opens up multiple possibilities for further research.

BIRD is a routing daemon developed in CZ.NIC research laboratories. It is, like HelenOS, an open source project. Porting of this application will require significant improvements of HelenOS network stack, possibly changes in operating system dependent parts of BIRD and creating a testing environment.

A machine running HelenOS with ported BIRD will become a router capable of dynamically configuring itself based on information it receives from other routers in the network.

The improvements made to the network stack will be reusable both by HelenOS native applications and future ported POSIX applications.

Content

The first Chapter describes BIRD on UNIX-like systems with emphasis on system dependent parts.

The second Chapter briefly covers HelenOS parts relevant to this thesis. That is mostly network stack and libraries related to it.

The third Chapter contains analysis. It describes prerequisites, possible approaches to porting BIRD and port requirements on the network stack and BIRD.

The fourth Chapter describes the implementation of changes in the HelenOS network stack, HelenOS libraries and a BIRD system-dependent layer.

The fifth Chapter evaluates the whole work by describing a series of tests in a virtual environment, proving that implementation is functional.

1. BIRD

BIRD is a routing daemon continually developed in CZ.NIC research laboratories. Currently, it is ported to Linux and BSD. It is deployed on some of the important Internet nodes, for example in Moscow, Milan, London.

BIRD's main role in a routing process is to keep the operating system routing table or tables updated according to information received from other routers in the network.

The most important functionalities that BIRD requires from an operating system to achieve this goal are network interfaces scanning, routing tables scanning, writing entries into routing tables, sending and receiving TCP, UDP and IP messages.

The following text first briefly describes BIRD's overall structure, execution flow, then focuses on OS dependent parts. Relevant protocol implementations are briefly described at the end of the Chapter.

1.1 Version

A new version of BIRD is released approximately every three to four months. The version ported to HelenOS is 1.5.0, which was the actual version of BIRD when work on this thesis started. The whole text of the thesis refers to this version. Documentation corresponding to it can be found archived at [6] and [7].

1.2 BIRD modules

BIRD splits its functionality into seven main modules. The core of BIRD is implemented in a module called `nest`. It contains structures and functions for storing and handling all data collected by BIRD. They can be obtained during synchronization with an OS (e.g. network interfaces information, local OS routes) or by one of the routing protocols (e.g. routes advertised by other routers, neighbors information).

The module `sysdep` contains a system dependent code. A separate Section 1.5 is dedicated to this module.

Routing protocols are implemented in the `proto` module. Currently, they are OSPF, RIP, BGP, RADV, static and pipe. Description of protocol implementations relevant to this thesis can be found in the Section 1.6.

BIRD uses a configuration file to configure routing protocols, kernel protocols, logging, filters and other options. The module `conf` handles parsing and interpretation of this file.

Filters are part of a simple, BIRD specific, programming language. They can be used to apply additional rules when passing routes between protocols and the core routing table. They are implemented in the `filter` module.

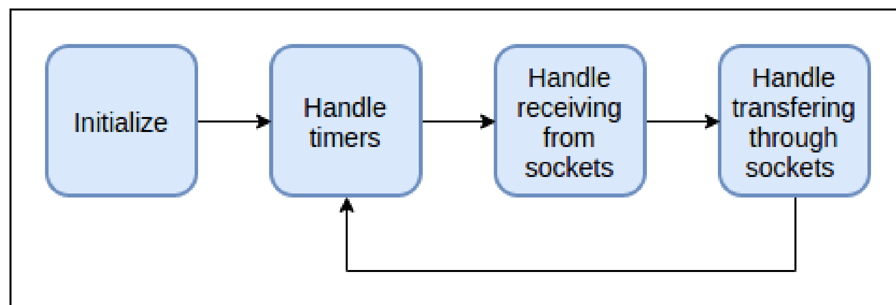
Helper functions, for example handling BIRD memory management, bit operations, IP address manipulation, checksum calculation are implemented in the `lib` module.

The last to mention is the `client` module. The BIRD client, which is a separate binary file, is compiled from this module's code. The client can be used for communication with a running instance of BIRD. It can display information about BIRD status, running protocols, protocol interfaces, routes, etc. It is also possible to reconfigure BIRD with another configuration file through the client.

1.3 Execution flow

Most of the application runs in a single thread inside the main loop. There is one exception, covered in the Section 1.4 . The Figure 1.1 shows key points of the application execution.

Figure 1.1: Bird execution flow.



Initialization

BIRD initializes modules and resources, performs a test for other running instance of BIRD, sets user ID and group ID and daemonizes process by forking it during this phase. The most important part of this phase is reading and parsing configuration for the first time. Protocols are initialized according to the configuration. Sockets are created and configured, timers for protocol-specific events are started and receive/transfer hooks are set during the protocol initialization.

Timer handling

The main loop is entered after the initialization phase. Time is updated at the beginning of the loop. The timers are examined next. Hook functions are executed on the ones that expired. Most operations are realized at this point. The operations are synchronization with OS routing tables and interfaces, protocol message transfer through sockets, rereading of the configuration file, etc. Timers are mostly recurring, so after executing the hook, they are rescheduled.

Socket reading and writing

BIRD executes `select` on two sets of sockets when timers are handled. Initially, the first set contains sockets with receive hooks, the second set contains sockets with transfer hooks. One socket can belong to both sets. After the `select`, the first set contains sockets that are available for reading without blocking, the second set contains sockets that are available for writing without blocking.

All sockets used by BIRD are configured to be non-blocking during initialization, which means reading a socket that has no available data returns error immediately.

If available, the data are received through the socket and passed to the receive hook associated with the socket. The receive hook processes the data according to the protocol and retrieved information is propagated to the core and possibly to other running protocols.

The data can be, for example, a protocol-specific message from a remote router, a message with a route or an interface information from OS.

The result of processing can be a new route in BIRD core, a new interface core structure, etc.

Protocols are sending messages in predefined intervals (timers are used). Only TCP sockets are checked for writing availability to determine if a socket is connected.

1.4 Threads

The BIRD documentation [6] is slightly inconsistent on this topic.

The design goals Chapter and the task documentation Chapter are stating that BIRD is a single threaded application. The main reason given by the documentation for choosing this approach is the complexity of locking mechanisms of a multi-threaded application. Custom scheduling mechanism is implemented to make BIRD responsive in real time. Bigger tasks are split into smaller parts and linked together with events and timers.

On the contrary, BFD Chapter states the protocol uses a separate thread. The protocol uses it to avoid being blocked by some of the bigger events for too long. The separate thread is only needed for the core part of the protocol. The rest of it runs in the main thread.

After examining the source code, it can be concluded that BIRD runs in a single thread with the exception of BFD protocol. This protocol uses POSIX thread library. Usage of this library is not separated into the system dependent module. It is used directly in the BFD protocol code.

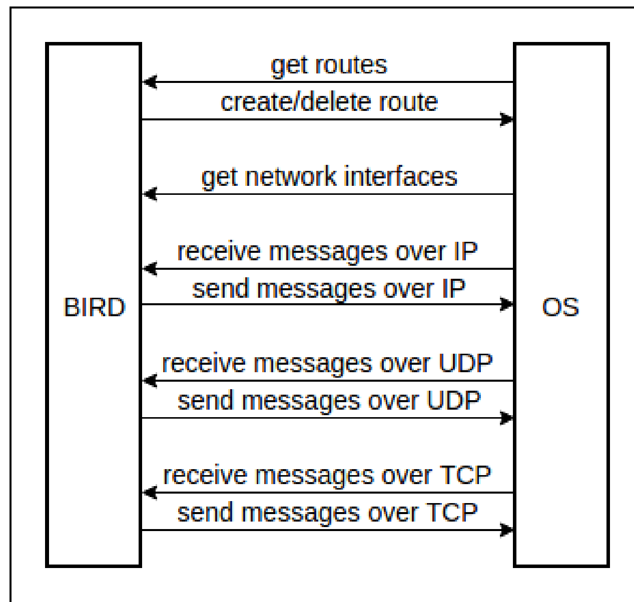
1.5 System dependent parts

BIRD tries to isolate all system dependent parts of the code into the `sysdep` module. There are some exceptions, for example, the BFD protocol. The module is further split into two layers.

- the upper layer, called UNIX, is a code common for all UNIX-like systems
- the lower layer is the OS family specific code. The implementation is chosen based on the target OS. It is configured before compiling. There are two implementations of the lower layer.
  - `linux` - The Linux OS family specific code.
  - `BSD` - The BSD OS family specific code.

The Figure 1.2 shows the most relevant exchanges between the OS and BIRD.

Figure 1.2: BIRD OS communication.



1.5.1 UNIX

This layer implements the main function and the main loop described in the Section 1.3. Also, functions for handling sockets, files, timers and the synchronization with the OS are implemented here.

Network sockets are used to provide routing protocols with means to send and receive messages over IP, UDP and TCP. UNIX sockets on this layer are used to check for an already running instance of BIRD and for communication with the BIRD client.

The code dealing with an OS synchronization is implemented by two protocols, `proto_unix_kernel` and `proto_unix_iface`. The first is for the routing table synchronization and the second is for the interface synchronization. They are represented by the same structures with hooks for events like a protocol start, reconfiguration, pre-configuration, post-configuration, shutdown, etc., as routing protocols.

The OS synchronization code here does not actually use any OS library. It only uses the lower layer (Linux or BSD). It expects functions for the interface scanning, routing table scanning and route replacement from the lower layer. Their implementation for Linux is described in 1.5.2 and for BSD in 1.5.3. All the calls to the lower layer are handled by timers created during the initialization or the reconfiguration of the two protocols.

Time updates can be handled in two ways. The monotonic clock implementation is used if provided by the OS. Otherwise, BIRD's internal clock implementation is used.

1.5.2 Linux

The Linux layer implements the synchronization with the OS using kernel sockets. These sockets, called RTNETLINK, have `PF_NETLINK` domain, `SOCK_RAW` type and

NETLINK_ROUTE protocol. RTNETLINK sockets are specific for Linux. Three of them are opened during initialization. One for scanning interfaces and routes, one for sending requests (deleting routes) and the last for an asynchronous communication with the OS.

Four most important functions used by the upper layer for handling the synchronization with the OS are described next.

- `void kif_do_scan(struct kif_proto *p UNUSED)` - Interface scanning. A message requesting an interface dump is sent to RTNETLINK socket. The socket is then read for messages until a special message denoting the end of data is returned. Each returned message contains information about one network link.

  Attributes, like interface index, name, MTU, flags are parsed from the messages for each link. This information is passed to BIRD's core, using a function for interface update.

- `void krt_do_scan(struct krt_proto *p UNUSED)` - Routing table scanning. The same principle as with the interface scanning is used. A request is sent to the socket and then messages containing routes information are read until the end of data.

  Route attributes like a destination network, a mask, a gateway, an origin (for example an user added route, a route added during boot, a BIRD added route) are parsed from received messages and passed to the BIRD's core.

- `void krt_replace_rte(struct krt_proto *p, net *n, rte *new, rte *old, struct ea_list *eattrs)` - Route replacement. Two routes are passed to this function. The old route is deleted from the OS routing table and the new route is added to it. If the old route is null, only the new route is created. Likewise, if the new route is null, only the old route is deleted. Routes are created and deleted by sending a message with specific route attributes to RTNETLINK socket.

- `nl_async_hook(sock *sk, int size UNUSED) -` asynchronous hook to RTNETLINK socket. The hook is called when kernel sends data to RTNETLINK socket without a previous request. This can happen, for example, when a user creates or deletes a route. The kernel sends a message containing an information about the route then. A separate RTNETLINK socket is used for receiving these messages.

Other functions implemented here worth mentioning are functions for setting socket options like MTU, multicast, MD5 authentication and parsing of structures containing messages and ancillary data from sockets.

1.5.3 BSD

The BSD system dependent layer utilizes `sysctl` system calls and one kernel socket with `PF_ROUTE` domain , `SOCK_RAW` type and `AF_UNSPEC` protocol to synchronize with the OS.

- `void kif_do_scan(struct kif_proto *p UNUSED)` - An interface scanning.
  Uses the `sysctl` system call. Management information base (array of integers) with six values is passed as first parameter. Values of this array are set from the top level (index zero) to the bottom level (index five) to `CTL_NET`, `PF_ROUTE`, 0, `AF_INET`, `NET_RT_IFLIST`, 0. The `sysctl` is called twice. The first call acquires the size of an output buffer. The buffer is allocated and passed to the next call. After the call, the buffer contains messages with information about all interfaces. The messages are parsed and the retrieved data are propagated to the BIRD core.

- `void krt_do_scan(struct krt_proto *p UNUSED)` - A routing table scanning. Uses `sysctl` utility similarly to interface scanning. The difference is in the fifth value of passed management information base, which is set to `NET_RT_DUMP`. The output buffer will now contain messages with route information.

- `void krt_replace_rte(struct krt_proto *p, net *n, rte *new, rte *old, struct ea_list *eattrs)` - A route replacement. The function sends a message with a route to the kernel socket. All attributes needed to create or delete route are stored in `rt_msghdr` structure. Attribute `rtm_type` of this structure is set to `RTM_ADD` or `RTM_DELETE`.

- `krt_sock_hook(sock *sk, int size UNUSED)` - The kernel socket asynchronous hook. Kernel sends to it information about deleted/created routes, same as on Linux.

1.6 Protocols

The Section describes protocols that are functional on HelenOS after the port. Overview of all three of them can be found both in BIRD documentation [7] and respective Wikipedia pages [14], [15], [16].

1.6.1 Border Gateway Protocol

The protocol is designed to be used between routers representing autonomous systems. The underlying protocol of BGP is TCP.

BIRD creates one BGP instance for each BGP connection. There are initially created two sockets for each instance - a passive TCP listener socket and a socket for initiating connection. The socket initiating connection is periodically recreated afterwards. The TCP connection can be established by either of them, in which case, the listener is closed and connection is stopped from being reinitiated.

The BGP exchange starts with sending and receiving an open packet. Routes are sent in update packets. Other packets used by BGP are notification, route refresh and close. Received routes are propagated to BIRD's core.

1.6.2 Open Shortest Path First

OSPF is implemented directly over IP. The only additions to OSPF packet are IP and Ethernet headers. BIRD uses sockets with AF_INET domain, SOCK_RAW type and protocol 89 (OSPF protocol number according to Internet Assigned Numbers Authority [1]) to send and receive OSPF packets. When BIRD is configured to use OSPF protocol on a network interface, it creates one raw socket for each of the interface addresses (if there are multiple addresses assigned to one interface, one socket is created for each).

The protocol is initialized when a hello packet is received and eventually the two routers exchange their graph representations of the network. Each makes necessary updates based on received information. Network graphs should be identical on both routers at this point.

BIRD recalculates routes from the updated network graph using Dijkstra's algorithm and sends updates to the core.

If one router does not hear from the other for a configured period of time, it expects that the neighbor is dead. In this case, the protocol state is set to down (the protocol is restarted) and the graph is updated.

1.6.3 Routing Information Protocol

RIP is implemented over UDP. BIRD uses sockets with AF_INET domain, SOCK_DGRAM type and protocol IPPROTO_UDP to send and receive data of this protocol. It creates one socket on each configured interface, then listens and periodically sends RIP packets. Implementation in ported version of BIRD does not start communication by request message as it should. Instead it starts directly advertising its routing table. This may lead to problems covered in the evaluation Chapter 5.

2. HelenOS

HelenOS is an OS with microkernel architecture. All the functionality, including network stack, is provided to the user space applications by separate modules called servers. An API for convenient usage of the services can be found in HelenOS libraries. It was not one of the design goals to make this API POSIX compliant. Despite of it, some parts of the API are kept in compliance. Unfortunately, the network stack is not one of them.

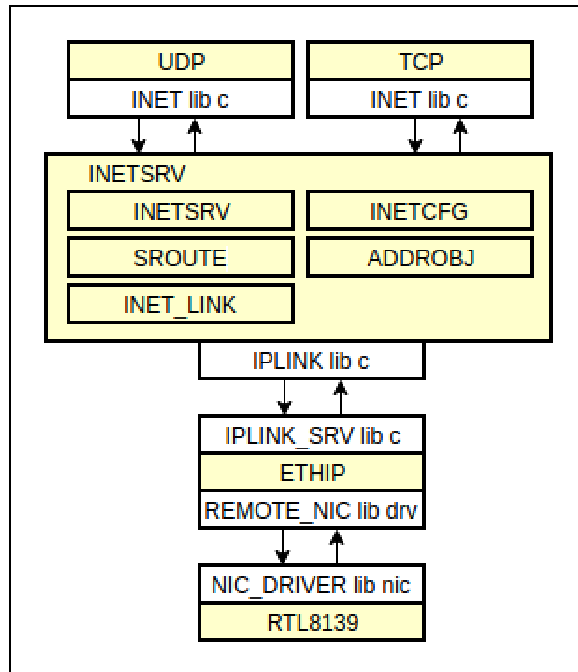
The network stack is an essential component of HelenOS for this thesis. Therefore, this Chapter is dedicated to it. The HelenOS coastline is mentioned at end of the Chapter.

2.1 Network stack

The HelenOS network stack consists of multiple servers. Servers communicate between themselves and with other applications through IPC messages. Upper layer servers are acting as clients of lower layer servers. Each server exposes one or more services for handling client IPC requests. Set of library functions for issuing these request, passing parameters and acquiring return values is prepared for each service.

Services are usually registered during the server initialization. Service IDs can be looked up by a name or a category. When a service ID is acquired, the client starts the IPC communication and obtains a session by connecting to the service. A callback is registered next. Servers use callbacks to pass IPC messages to clients asynchronously. Network stack servers use it to pass received network messages, making the whole receiving part of the network stack asynchronous.

Figure 2.1: HelenOS network stack.



The Figure 2.1 shows network stack servers and relations between them. Only servers relevant to BIRD port are displayed. The servers are displayed in yellow color. White boxes attached to them are most relevant libraries used for communication with other servers or applications.

Network interfaces are referred to as links in HelenOS network stack code. The text in this Chapter will refer to network interfaces as links in some cases for this reason. Following Sections describe the stack top to bottom.

There are described unmet requirements for IPv4 routers by the particular layer at the end of each Section. The requirements are specified in RFC 1812 [5] directly or indirectly by reference to another RFC document. The indirect references can be found in the previous document. The network stack from mainline with routing implementation from [3] and no further modifications is considered. It is needed to mention that HelenOS meets only very few of the requirements.

2.2 Transport layer

There is one server for each of the two transport layer protocols, UDP and TCP.

2.2.1 TCP server and library

A client starts using the TCP server by creating a TCP instance. The server creates a callback for the instance during the call. The client can now use the instance to create a TCP connection or a TCP listener.

A client can create a new connection by specifying an Internet endpoint pair (a local address, a local port, a local link, a remote address, a remote port) and a set of functions invoked by the callback when an event related to the connection occurs. The local address and the port can be omitted, in which case they will be assigned automatically. The functions are for handling following events: a connection established, a connection failed, a connection reset, data available, urgent data. A structure representing the connection is created both on the server and the client.

The TCP listener is created by specifying an Internet endpoint (a local address and a port to listen on for new connections), a function handling new connections and a set of functions for each connection as described in the previous paragraph.

Notice that when the client creates the connection, it must be closed explicitly. The life cycle of an incoming connection is handled by the callback. When the function handling a new connection returns, the connection is destroyed.

Data can be sent and received in a standard way (specifying pointer to a buffer and its size) once the connection is established.

2.2.2 UDP server and library

The UDP initialization is very similar to the TCP initialization. The client creates a UDP instance and the server creates a callback. A UDP association needs to be created next. The association is uniquely identified by an Internet endpoint pair. Also, a structure with three functions for handling callback events is added to each UDP association. The events are: a message received, an error message received,

a link state change. Both the server and the client are keeping a structure for each association.

The client can start sending and receiving messages when the association is created. Data, data length and possibly remote endpoint pair are passed to UDP server when a message is sent. The server creates PDU based on passed parameters and the association Internet endpoint pair. The source address, the source port and the link are given by the association. The destination address and port are given by a remote endpoint from the argument. If the passed endpoint is NULL, they are also given by the association. Finally, the UDP server converts the PDU into a datagram and passes it to the `inetsrv` server.

The UDP implementation allows only an asynchronous reading. When a datagram is received from the `inetsrv`, it is converted into a PDU. The client that receives the data is determined by the link, the local address, the local port, the remote address and the remote port. The clients association must match all these parameters in order to receive the PDU. The registered callback function on the client side receives only information about the message. It is up to the implementation of this function to retrieve message data from the server or ignore it.

2.2.3 RFC compatibility

There were not found any critical violations of basic RFC specification in UDP and TCP implementations. Additional requirements are however not met, mostly due to lack of support from the network layer. No IP options can be specified. TTL is not configurable. There is no interface to configure keep-alive behavior. TOS cannot be specified. There is no reaction to ICMP errors.

2.3 Network layer

Implemented by the `intesrv` server. This server contains most of the network stack logic. The logic is split into three services: the default service, the configuration service and the ping service. The default and configuration services are discussed next.

2.3.1 Default service

The service handles sending and receiving network messages. It is also responsible for routing. The service decides if a received message is passed to an upper layer, discarded or routed (re-sent using a lower layer).

A connection between a client and the default service must be initialized first. A client passes a protocol number to the service during initialization. It will receive all datagrams unpacked from IP packets with this number in IP header afterwards. This protocol number will be also assigned to the IP header of messages sent by the client.

The `inetsrv` server expects only one client per protocol, which limits the network stack. Subscription of multiple clients to the same protocol is not explicitly prohibited but results in a non-functional network stack.

Currently, there are two upper layer servers (TCP and UDP) connected to `intesrv` server. If some other server or application chooses to use `inetsrv` for TCP or UDP communication, it will either make the respective server not receive any datagrams or will not receive any datagrams itself.

When a client wants to send data, it passes a datagram to the `inetsrv` server. Datagram consists of a link service ID, a source address, a destination address, a type of service, a pointer to data and a data size. If the link service ID is specified, the datagram is sent directly to that link, regardless of the source or the destination address. Otherwise, the destination address is used to determine the link. It is compared to a list of interface network addresses first. If a match is not found, the routing table is consulted. In case a router (gateway) address for the given destination is found, it still needs to be resolved into an interface. Therefore, it is compared to list of interface network addresses. If the link is determined by this process, the datagram is packed into a packet and passed to the `ethip` service.

The default service provides a lookup of an interface address based on a destination address. An interface, through which the destination address can be reached, is looked up based on interface addresses and static routes. An address of this interface is returned.

2.3.2 Configuration service

The service is used to get and change the network configuration. The user space application `inet` provides command line interface to this service. The HelenOS network configuration is given by following parameters.

- Network interfaces (links) - the service holds a structure for each link. Attributes of the structure are a service ID, a service name, a session, a pointer to IP link, a default MTU, a MAC address and a MAC address validity flag. Links are discovered by the `nconfsrv` server as services of `ethip` and `loopip` servers on the network layer. When a link is discovered, the `nconfsrv` passes its service ID to the configuration service. At this point, `inetsrv` acquires a session with the `ethip` service, using the passed ID, and gets all other attributes of the link from it. Links cannot be added or deleted by a user.

- Network interface addresses - attributes of each are an ID, a network address (IPv4 or IPv6 address plus prefix), a pointer to a link and a name.

  When a client creates a network interface address, it is associated with a link according to the passed name. The service ID of the link is used to pass the address to the `ethip` server (link layer).

  When a client deletes a network interface address, it is only removed from the list in the `inetsrv` server. The address remains assigned on the link layer. This behavior is most likely not intentional. It affects the ARP protocol, which will respond with a MAC address even when asked about an address that was previously deleted on the network layer.

- Static routes - represented by structures with following attributes: a destination network address, a router (gateway) address, a name.

Interfaces, interface addresses and static routes are stored in linked lists. All three lists are handled the same way. A new link, a link address or a static route is always appended at the end of the respective list. When a client wants to delete a link address or a static route, it passes its ID. The list is iterated and if the ID is found, the item is deleted. Listing links, link addresses and static routes is described next. A client must obtain a list of IDs from the respective linked list first. The ID list is used to acquire items from the service one by one. It needs to be taken into consideration that the list is not locked on the server between getting IDs and getting individual items.

2.3.3 RFC compatibility

There were not found any critical violations of RFC requirements on basic IP implementation. The extensions are however not implemented. There is no support for subnetting and classless interdomain routing (CIDR). Broadcast and multicast addresses are treated same as standard addresses, both when delivered locally and when forwarded (the only exception is broadcast to all hosts, 255.255.255.255, treated as a local address). There is no path MTU discovery implemented. ICMP protocol sends/receives only ICMP echo request/response messages, meaning no other query (i.e. timestamp, address mask ...) and error (i.e. destination unreachable, redirect ...) messages can be sent or processed. IGMP protocol is not implemented. All IP header options are ignored on both incoming and outgoing packets. The destination of forwarded packets is decided based on packet destination address only, everything else is ignored (TOS, metric, IP header options, ...). The forwarding algorithm always selects the first route from the routing table, that prefix-matches the destination of a packet. The algorithm satisfies the basic rule on forwarding algorithms but violates the longest match rule. The packet filter is missing. The precedence field of the IP header is not processed in any way.

2.4 Link layer

The link layer in HelenoOS consists of NIC drivers and the `ethip` server.

Drivers for multiple NIC models are implemented in HelenOS (The Figure 2.1 shows the RTL8139 driver but it can be any of the implemented drivers). All of them are accessed through the same API. The API is defined in `nic` library. Drivers handle the communication between a hardware and the rest of the network stack. The driver can enable or disable multicast for its NIC, which is important for porting BIRD. Multicast is disabled by default.

The `ethip` server is responsible for a NIC discovery and passing messages between the network layer and drivers of discovered NICs. Additionally, ARP protocol is implemented here. The translation of multicast addresses by the ARP protocol is relevant for this thesis.

2.4.1 RFC compatibility

There is no mechanism to flush out-of-date cache entries neither a mechanism to prevent ARP flooding. HelenOS accepts broadcast and multicast addresses as

ARP replies. Interface MTU is not configurable. Link control protocol (LCP) is not implemented.

2.5 Coastline

HelenOS coastline is a tool for installing POSIX applications into HelenOS. It is stored in a separate repository. Its central part is the `hsct.sh` script. The usage is as follows. A build directory needs to be initialized first. The initialization includes copying of HelenOS header files and libraries. A HARBOUR file must be created for the application next. Most commonly, the HARBOUR file specifies an URL with sources of the ported application, files for patching and scripts to configure, make and package the application. The application is ready to be installed when build directory is initialized and HARBOUR file is created, .

3. Analysis

The Chapter describes overall approach to porting BIRD. A subset of BIRD functionality to achieve in HelenOS is proposed based on the dependencies, current HelenOS state and testing environment capabilities. Changes in HelenOS and BIRD required to achieve proposed functionality are analyzed afterwards.

3.1 Testing environment

The very first goal during the work on this thesis was to get familiar with BIRD. The next step after going through the official documentation and briefly examining the source code was to get a hands-on experience with a running instance.

It is possible to compile and run BIRD on a local machine but without a network with other participants there is not much to observe.

3.1.1 Virtual vs real hardware environment

There were two possible approaches on how to put BIRD into context - a real hardware network and a virtual network. The virtual environment was an obvious choice here. Real hardware does not provide any significant advantages.

Advantages of the virtual environment are, for example, great scalability (simple adding and removing of network nodes and connections), clean environment on each rerun and it is much more cost-effective, faster to set up, accessible.

3.1.2 Choice of software for network simulation

It was expected that there already existed a virtual environment created during the BIRD development. Surprisingly this was not the case. The environment had to be created from scratch.

Possibilities of graphical software for network simulation (GNS3, CORE, Marionnet, etc.) were examined first. The chosen tool will be used throughout the whole development process. Key requirements for the software are listed next.

- Capability to connect QEMU VM's running both UNIX and HelenOS.
- Provide means to simulate other network components, at least switches.
- A fast environment startup, stable during execution.

Unfortunately, none of the tested tools met all the requirements. Most of them became unstable/slow after HelenOS was plugged in. For example, GNS3 expects OS to be installed on hard drive. When a workaround was used to pass a QEMU instance with an OS booted from an image, the network stopped working.

This led to VDE be the tool for virtualizing the network. VDE meets all the requirements. The only downside is no graphical representation of the network. Use of this tool was inspired by the thesis HelenOS packet filter [3].

3.1.3 Virtual network participants

It was decided that the network will be comprised of VMs running BIRD only. Use of different routing software would complicate the development process unnecessarily. It is much simpler to follow a debug output and to debug routing protocols on multiple machines simultaneously when all routers in the network are running BIRD. There is no need to get familiar with another routing software.

An OS running on VMs was needed to be chosen next. BIRD was ported to BSD and Linux systems, so the task was to choose a flavor from one of them.

The Core distribution of Linux in its most basic version with only command line seemed to be the perfect fit. It is one of the smallest distributions available. Reasons behind this choice were as follows.

- fast to boot - this is important because rebooting will be happening very often during the testing.
- a small installation size - the smaller installation, the easier it is to manage in a version control system.
- whole OS runs in memory - all changes, including network settings and file system changes, are lost on reboot unless explicitly saved. It ensures the same starting state for testing on each rerun.

3.1.4 Running BIRD

The first virtual environment consisted of two Cores connected with the VDE. They were able to communicate when the network was configured.

At this point, it was simple to configure different protocols with different options and observe the behavior of BIRD on both machines by following a debug output and the content of routing tables.

Virtual environment was easy to extend for later stages by copying core with BIRD, changing configuration and connecting it to the network using VDE.

A long-term strategy was to achieve a state in which one of the Cores would be swapped for HelenOS with BIRD. The behavior of the network should remain the same considering the routing.

3.2 Porting BIRD

A coastline build revealed all missing OS dependencies in HelenOS because BIRD was compiled and linked against HelenOS header files and libraries. BIRD was configured for a cross-compilation. The target OS was configured to be Linux, so the Linux system-dependent layer was used during the compilation.

3.2.1 Missing dependencies

Header files not found by BIRD during compilation are listed next. Almost all functions, macros and structures required by BIRD but missing in HelenOS were declared/defined in these header files. A brief description of them is included.

`linux/if.h` - network interface flags. Used to parse messages containing interface information from netlink sockets.

`linux/netlink.h` - netlink socket address structure, netlink message and error message structures, macros handling netlink messages. The structures are passed to socket API to exchange information about routes and interfaces between BIRD and an OS.

`linux/rtnetlink.h` - macros and flags for parsing interfaces and routes received from the kernel through netlink sockets.

`net/if.h` - structure for interface names. Passed when binding socket to interface.

`netinet/in.h` - socket address and packet information structures, protocol enumeration and macros defining socket options. The packet information structure is one of socket control messages.

`netinet/tcp.h` - a socket option level macro for TCP.

`netinet/udp.h` - actually not used, redundant dependency.

`netinet/icmp6.h` - macros and structures for ICMP over IPv6 filtering.

`sys/socket.h` - the most essential header containing socket API, macros defining socket option layers, socket option names, socket domains and socket types. The important structures defined here are common socket address structure, socket address storage structure, control message structure and socket message structure.

`sys/select.h` - select function and macros for manipulating sets. Used to determine socket read/write availability.

`sys/time.h` - time types, already defined by HelenOS in `time.h`.

`sys/uio.h` - input/output vector structure. An array of vectors is a part of a socket message.

`sys/un.h` - UNIX socket address structure. Used to create a kernel socket with specific address checked when BIRD is started to ensure only one instance of BIRD is running.

`alloca.h` - alloca function.

`glob.h` - glob function, structure and macros this function operates with.

`grp.h` - setsid, setgid, chown and getgrname functions and group structure for chown function. Used during BIRD initialization on UNIX.

`libgen.h` - dirname function.

`termios.h` - tcgetattr, tcsetattr functions, termios structure and macros used by these functions. Only included by BIRD client.

BIRD has other OS dependencies (stdio.h, stdlib.h, unistd.h, etc.) already present in HelenOS. These will not be covered here.

3.2.2 Mocking

Two possible approaches could have been taken from here.

The first one was to decide which dependencies will be implemented in HelenOS and which will be replaced by HelenOS alternatives right away.

Another approach was to create mockups of all the dependencies and then replace or implement them continually one by one.

The second approach was taken. The advantage was that BIRD could have been compiled and installed into HelenOS quickly. A downside was that some time would be spent on the creation of mocks and header files, that could be potentially deleted later.

Almost all mocked macros and structures are related to mocked functions. Structures are passed as parameters and macros are used for preparing an input or parsing an output. Missing functions that needed to be mocked are listed in Figures 3.1 and 3.2. This shows that all the networking and OS synchronization logic is hidden behind socket API.

Figure 3.1: Socket API.

```
int socket(int, int, int);
int setsockopt(int sockfd, int level, int optname,
    const void *optval, socklen_t optlen)
int bind(int, const struct sockaddr *, socklen_t);
ssize_t recvmsg(int, struct msghdr *, int);
ssize_t sendto(int, const void *, size_t, int,
    const struct sockaddr *, socklen_t);
int connect(int, const struct sockaddr *, socklen_t);
ssize_t sendmsg(int, const struct msghdr *, int);
int listen(int, int);
int getsockname(int, struct sockaddr *, socklen_t *);
int accept(int, struct sockaddr *, socklen_t *);
```

Figure 3.2: Other API.

```
int select(int, fd_set *, fd_set *, fd_set *, struct timeval *);
void *alloca(size_t);
int glob(const char *, int, int (*errfunc) (const char *, int),
    glob_t *);
void globfree(glob_t *);
```

3.2.3 BIRD features to be ported

Each protocol and each feature has its own set of requirements on sockets and other OS-specific functionality. Implementing all of them is beyond scope of this thesis, so a reasonable subset had to be chosen. Reasoning behind choices of implemented features follows. Features functional with mocked dependencies are not mentioned (e.g. configuration file parsing).

- Kernel and device - protocols responsible for synchronization between BIRD and OS. No other protocols can function without device and kernel protocols. Support is essential.

- OSPF - the most popular interior gateway protocol. Porting and testing of this protocol is prioritised.

- RIP - even though usage of the RIP protocol in networks nowadays is rare, it does not require any network unrelated dependencies. The protocol will be included in ported features.
- BGP - a protocol for exchanging routing information between autonomous systems. The protocol will allow HelenOS to act as a backbone router. Protocol will be supported.
- BFD - not a routing protocol by itself. BFD enhances routing protocols by detecting neighbors upstates. Detection is much faster than the one implemented by routing protocols themselves. Protocol requires POSIX threads, which are not implemented in HelenOS mainline. It will not be supported.
- IPv6 support - there are multiple reasons why supporting IPv6 would be difficult. The HelenOS network stack does not support sending IPv6 messages directly into a specified link, which is essential for BIRD. As stated in [13], the IPv6 implementation support for routing and multiple NICs is very limited. HelenOS is not ready to be used as a router in an IPv6 environment with the current IPv6 implementation.
- BIRD client - an administrative tool that can be used to reconfigure running BIRD or print information about its state. There are two versions of this tool. The more advanced version requires the readline library which is missing in HelenOS. The lightweight version will be ported.

3.2.4 POSIX dependencies to be implemented

BIRD will have to be able to synchronize itself with HelenOS in a similar way it synchronizes with Linux or BSD to achieve the proposed functionality. Also, it must be able to use HelenOS network stack to send and receive messages over IP, UDP and TCP to support OSPF, RIP and BGP.

One way to fulfill the requirement is to implement mocked dependencies in HelenOS. Another way is to replace Linux and Unix dependencies with HelenOS native functionality in BIRD. It was necessary to decide which dependencies to replace and which to implement. Dependencies of unsupported functionalities will remain mocked.

Three main options presented themselves due to BIRD's design of system dependent parts.

- replace all BIRD system dependent parts with HelenOS native support.
- replace the lower system-dependent layer (Linux) with HelenOS native support and implement dependencies from the higher system-dependent layer (UNIX) in HelenOS.
- implement all dependencies in HelenOS.

The UNIX system-dependent layer counts around 4000 lines of code. The only dependency needed to make it functional in HelenOS are network sockets.

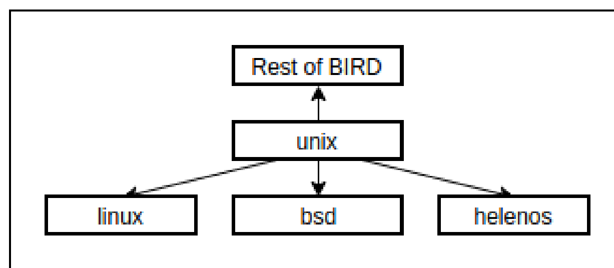
Moreover, BIRD's single-threaded design would not allow direct usage of HelenOS asynchronous network API which would have to be synchronized either in BIRD or in HelenOS. Since the synchronization is unavoidable, it can as well be implemented behind the socket API. Implementing sockets in HelenOS is a much simpler solution than rewriting the UNIX layer.

The Linux system-dependent layer requires RTNETLINK sockets. HelenOS already provides an API for getting interface information, getting routing table information, adding routes and deleting routes. The HelenOS API can be considered even more comprehensible and user-friendly than the RTNETLINK socket API. There is no need to convert the API from an asynchronous to a synchronous, unlike the network socket API. Conveniently, all the RTNETLINK socket functionality is isolated in BIRD's Linux layer. Additionally, the layer is much smaller than the UNIX layer and there is not much functionality that can be reused in HelenOS.

The final decision was to implement network sockets in HelenOS and create a new lower system-dependent layer for HelenOS in BIRD.

The Figure 3.3 shows BIRD system-dependent layer organization after this decision. The UNIX layer calls one of the lower layers (depends on the platform BIRD is compiled for). It also calls the rest of BIRD from the main loop.

Figure 3.3: BIRD layers regarding OS dependence.



3.3 HelenOS socket design

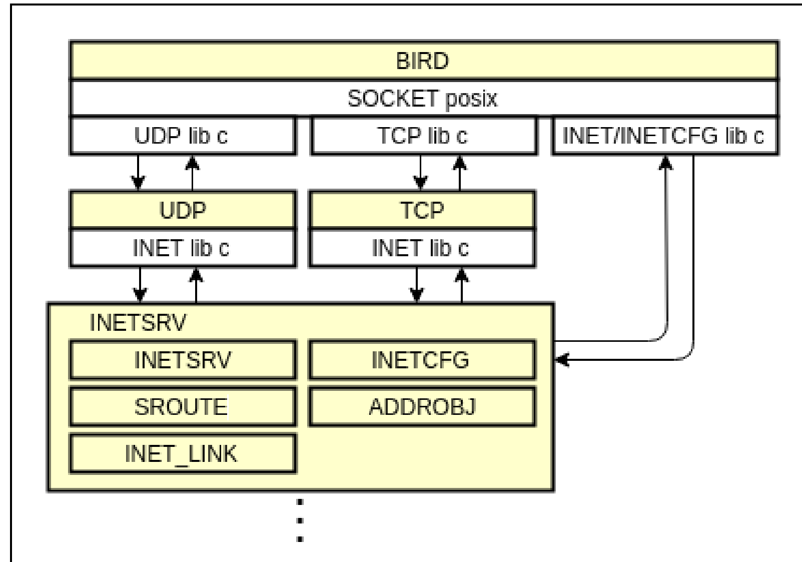
Multiple sources pointed to the fact that sockets were implemented in HelenOS in the past but were removed before work on this thesis started. The idea was to find a revision before their removal and see, what can be salvaged from there. After looking up the revision and examining the socket related code, the conclusion was that it was removed for a good reason and it is not salvageable due to its overall low quality. Sockets had to be designed from scratch.

The initial design was to implement sockets purely inside POSIX library. The POSIX socket library would use C library network stack API as shown in the Figure 3.4.

Soon enough, the code base inside the library became too big and the design did not conform to HelenOS microkernel multiserver architecture. Handling of socket file descriptors and resource locking inside library became problematic. This design would result in a library only usable by BIRD.

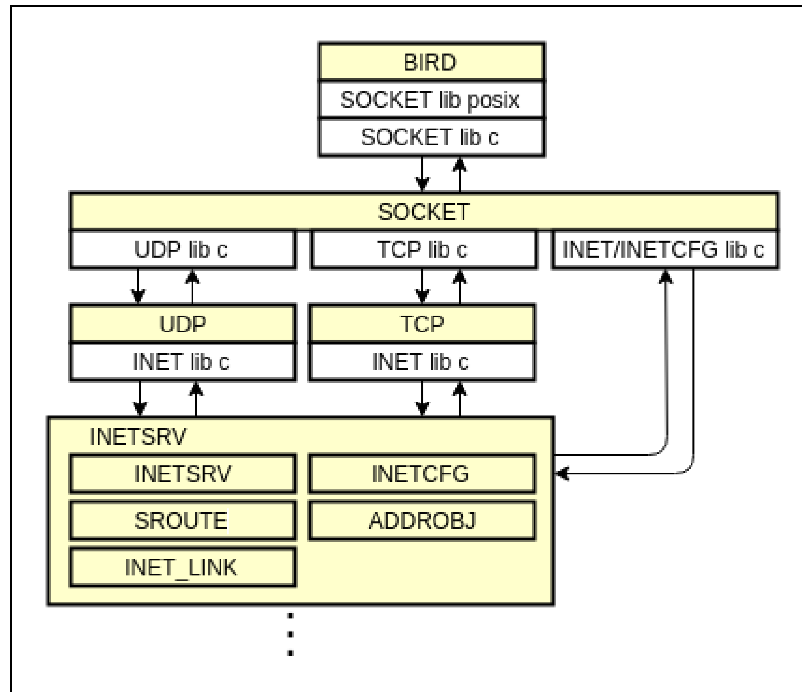
Socket logic had to be moved into a new server. The server would be accessed through a C socket library. POSIX socket library would be presented to BIRD

Figure 3.4: Initial socket design and its incorporation into network stack.



like before. The Figure 3.5 shows the final design and its incorporation into the network stack.

Figure 3.5: Final socket design and its incorporation into the network stack.



3.3.1 Design goals

Sockets are used for multiple purposes and can be configured and adjusted in many ways on UNIX systems. Socket implementation equivalent to UNIX sockets was neither realistic nor needed. With this in mind, following goals were set.

- sockets will be usable both by POSIX applications installed from coastline and HelenOS native applications.
- there will be no need to patch BIRD, except for the lowest system dependent layer and the BIRD client.
- only functionality needed by BIRD will be implemented but sockets will be easily extensible.

The first goal will be met by implementing socket API in C library and extending it into POSIX library. The C library will handle passing data between the user of the library and the socket server, no more logic will be implemented here.

The second goal will be achieved by emulating all the BIRD required functionality by the socket server.

To achieve the third goal, the design will follow this rule. All parameters passed to socket API will be sent to the socket server without any alteration, regardless of if they are used or not. The server will always have exact copies of passed parameters, including complex structures.

When all the data arrive to the server, it will be decided if a implementation of a function for that particular combination of received parameters exists. If it does, it will be called and all expected values will be returned. If not, it will notify the library about a missing implementation.

3.4 Sockets requirements

The mocked API from the Figure 3.1 had to be implemented. Each network socket used by BGP, OSPF or RIP is bound to an interface right after it is created. It is done by setting a socket option where an interface name is passed as a parameter. Sockets used by BGP and RIP are bound to a port additionally. It must be possible to bind sockets this way.

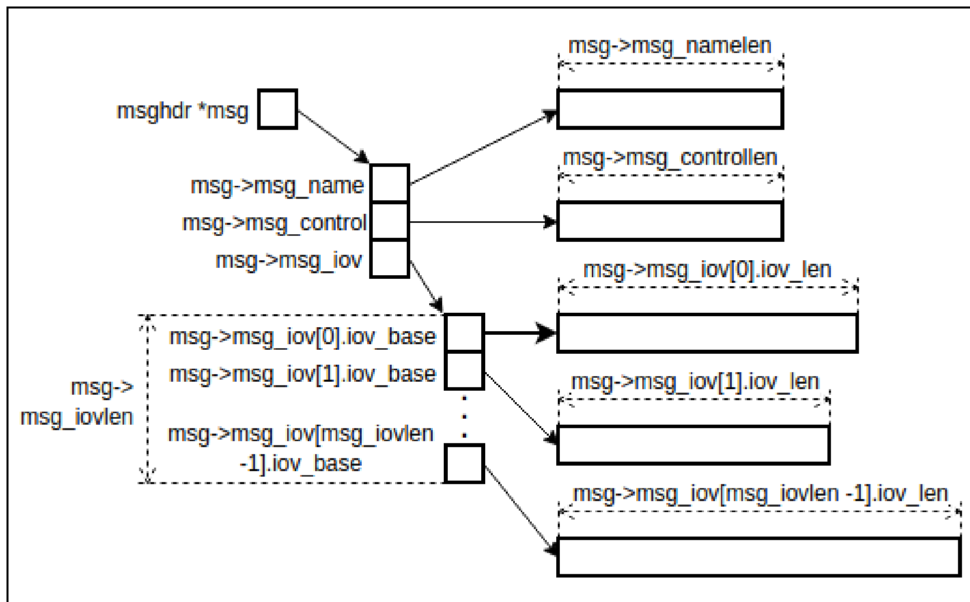
Reception of network messages for OSPF and RIP must be converted to synchronous. TCP receiving for BGP is already synchronous for BGP but the TCP connection initialization must be synchronized.

BGP is sending and receiving network messages by writing to a socket and reading a socket in a standard way. RIP and OSPF are using the `msghdr` structure both for sending and receiving network messages. Understanding of the structure is crucial to understand how the two protocols use OS to communicate with other network participants.

It consist of three complex parts plus `int msg_flags`. Flags are not used by any implemented protocol. The complex parts are described next. The structure is also illustrated by the Figure 3.6.

- `void *msg_name` - a buffer of size `msg_namelen`. Usually contains socket address describing message destination when sending or message source when receiving.
- `struct iovec *msg_iov` - a pointer to array of input/output vectors. The `msg_iovlen` denotes number of vectors in array. Each vector contains buffer

Figure 3.6: Message header structure.



void *iov_base and its size iov_len. The buffers either contain data to send or are filled with received data.

- msg_control - a buffer of size msg_controllen. It can be filled by a user with one or more control messages to further configure message sending. When receiving, it is filled by socket implementation and contains control messages with additional information about receiving.

The sending and the receiving will have to deal with translation between msghdr structure and form suitable for HelenOS network stack. The translation must respect the socket setup. This task will be handled by the socket server.

Only a non-blocking implementation of receiving messages and accepting connections is needed by BIRD. The select function and macros related to it must be functional for sockets. In particular, it must be able to create a subset of sockets with data available for reading from the read set. The read set can contain any mix of TCP, UDP, UNIX and raw sockets. The write set is used only for TCP sockets through which is initiated TCP connection. The select must be able to create a subset of sockets with established TCP connection from the write set. The exception set is not used.

3.5 Additional requirements

There are three main categories of additional requirements. Multiple requirements were found on lower network stack layers needed by sockets. Other changes in the network stack were needed for the routing. The last category of requirements is on BIRD itself. All the requirements are listed next.

Multicast

It must be possible to enable a multicast on a NIC associated with a particular link layer name due to both OSPF and RIP using muticast addresses. At the time, it was not possible to programmatically pair a link name (e.g. `net/eth1`) with NIC driver service (e.g. `devices/\hw\pci0\00:03.0\port0`) at the application layer.

The ARP protocol lacked support for multicast addresses so at least the ones used by BIRD must be properly translated by this protocol.

Deleted interface addresses and routes, route origin

BIRD scans for deleted routes and interfaces during synchronization. HelenOS must be able to keep track of interfaces and routes even when they are deleted. Routes are treated differently based on their origin. They can be installed on Linux by an administrator, during boot, by the system (when interface addresses are configured), by BIRD, etc. Route information must include the route origin.

Routing

There was an already existing implementation of routing created in the HelenOS packet filter thesis [3]. The thesis was never merged into the mainline repository, so the relevant parts must be imported. It was found out the imported routing mechanism can route only one hop. The routing must be functional for any number of hops.

3.5.1 Routing table data structure

The routing table is implemented as a linked list. The list must be replaced with a more suitable data structure. Except for linked list, the only data structure implemented in HelenOS user space is a hash table. An AVL tree and a B+ tree can be found in kernel code. None of the mentioned data structures is suitable for a routing table. The data structure must allow a prefix search in a reasonable time. UNIX systems are using a binary trie or one of its modification as a data structure for the routing table. A binary trie will be implemented in HelenOS.

3.5.2 Source address of outgoing UDP messages

If UDP association is created without a specified local address (in the local endpoint of the endpoint pair), the source address of messages sent through this association is null. BIRD only specifies a link through which the messages are sent and received. If an interface address is used as a local association address, the association receives only messages destined to that address (messages destined to multicast addresses are filtered out). Following setup must be possible for a UDP association. Messages destined to all addresses from a given link are received through the association and at the same time a local address is added to all outgoing messages.

3.5.3 Network stack bug fix

The network stack is failing when HelenOS is connected to Core UNIX through VDE. It happens every time HelenOS network is configured first and Core network second. When configuration happens in reversed order, failure occurs only occasionally. BIRD cannot be tested under such circumstances. The bug must be located and fixed. An initial version of the bug is filed under [8]. A version with a detailed description and a solution is filed under [9].

3.5.4 BIRD client

The client expects that the console is the standard input and uses the `select` function to wait for commands. HelenOS does not support select for VFS. The client needs to be redesigned slightly.

3.6 BIRD's HelenOS system dependent layer requirements

The four functions covered in 1.5 must be reimplemented using HelenOS native support. A route origin and a storing of deleted interface addresses and routes are prerequisites for the implementation. The layer must also define hooks like a reconfiguration, a start and a shutdown. The hooks are defined for compatibility reasons and will remain empty or contain just a very simple logic.

4. Implementation

This Chapter describes the implementation from the point when BIRD is compiled into HelenOS with mocked dependencies and HelenOS is running inside a virtual environment connected to a Core UNIX VM.

4.1 Debugging techniques

The only effective debugging techniques proved to be logs for servers and console outputs for libraries and BIRD. At the early stages, GDB was tried but turned out to be ineffective. Compilation time is multiple times longer when line debugging information is included. Values of user space registers EIP, ESP and EBP need to be found out and set in GDB to avoid collisions. An address of the `.text` section must be manually loaded to add symbol information. This process must be repeated on each restart of the testing environment.

Fortunately, this is only true for HelenOS. GDB can be installed and used to debug BIRD inside virtualized OS when it is running on the Core UNIX. The behavior of BIRD running on Core and incoming traffic from HelenOS can be observed this way.

Additionally, `tcpdump` utility was used in Core UNIX to monitor traffic coming from HelenOS.

4.2 Socket implementation prequisities

Only the final implementation is described. The phase in which logic resided inside the POSIX library is skipped. The implementation was continually tested using BIRD. When a socket of a certain type was under development, a protocol using this type was configured in BIRD.

There was no need to create redundant applications on HelenOS and Core for testing sockets. BIRD already did what was expected from such application - create and configure sockets, then start continually sending and receiving messages through them. An interference from the rest of the application did not impose any issues.

Network sockets were implemented first. BIRD cannot use sockets if the OS synchronization is missing, so the functions scanning interfaces and routing table were mocked to return predefined values. The values were matching the real state of HelenOS network configuration. Writing received routes into routing table was not needed to be functional to test sockets. This way, synchronization with OS was simulated and development could focus on sockets.

4.3 Socket POSIX library

The POSIX library contains only external references to the socket API in the C library. There are three exceptions.

Functions `close`, `read` and `write` are shared between VFS and sockets. The highest possible VFS file descriptor is 127. Sockets will use only higher file de-

scriptors. The POSIX library API can determine if the file descriptor belongs to a VFS file or a socket and call its C library implementation. The original C library `close`, `read` and `write` functions are called for VFS files and newly created `sockclose`, `sockwrite` and `sockread` functions are called for sockets.

4.4 Socket C library

The only socket logic implemented in the C library deals with passing and receiving data between the user of the library and the socket server through IPC. Also, the socket API documentation can be found here.

Before the client can start communicating with the server a session must be created. This is usually hidden behind some initialization library call. The standard approach is that client holds one session per server throughout its life cycle, even though it is possible to create multiple sessions with one server.

Socket library implementation uses one session to handle all sockets. This session is initialized when the first socket is created and is used for all subsequent calls.

Another option was to create one session for each socket, which would create an unnecessary overhead. Moreover, the sessions would have to be tied to socket file descriptors, meaning these descriptors would have to be implemented inside the library, which is undesirable.

4.4.1 Header files

The defined socket related structures and macros must be visible to the socket server, the C library and the POSIX library. Structures and macros related to unsupported functionality (mostly IPv6) are an exception. They can remain in the POSIX library as mocks.

All the header files containing at least one structure or macro used by the C library or the socket server are placed inside the C library. Header files with function declarations are split into a function header and a macro/structure header. Header files containing only macros and structure definitions are placed under types folder. They can be included from both libraries and servers.

Macro/structure headers are grouped under the socket folder (subfolder of types), even though POSIX applications expects them scattered in different places (e.g. netinet/in.h, net/if.h). The problem is solved by adding the same set of headers properly placed in the POSIX library. Each POSIX header includes its counterpart in types.

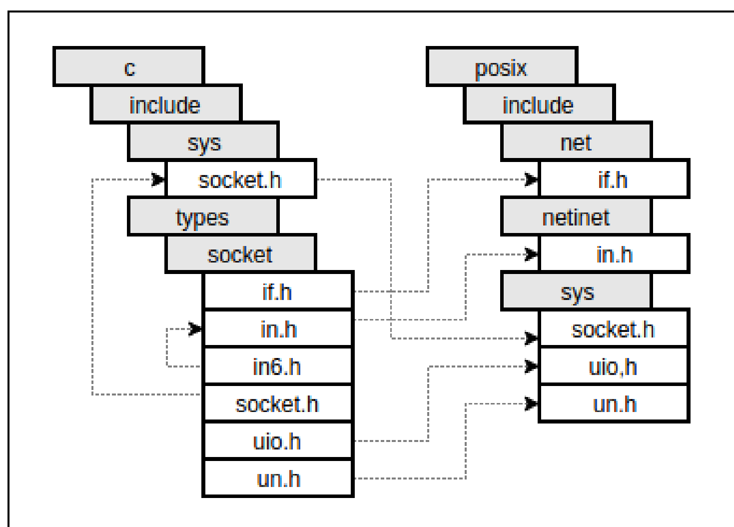
The Figure 4.1 shows source tree of most important POSIX and C library socket header files and their relations. Arrows mean inclusions.

The C library socket API is declared in `sys/socket.h`, same as on UNIX.

4.4.2 C library socket API implementation

Most of the socket functions are able to pass all the parameters in arguments of an initial IPC call plus in one or two more calls for passing large data. If any of the calls fail at some point, -1 is returned and `errno` is set to an appropriate

Figure 4.1: Hierarchical structure of socket related libraries.



error code. There are two functions that take `msghdr` structure as a parameter and therefore the communication is more complex.

The first is `ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags)`. It sends a socket file descriptor, a number of input/output vectors and flags in the initial IPC call. All parts of the message header are sent as large data in separate calls. The socket address is sent first. Input/output vectors follow, each in one call. The control message is sent is last.

The second is `ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)`. Parameters are same as for the `sendmsg` function, but here the data from the `msghdr` structure are not sent to the server. It is filled with data from the server instead. The server only needs to know dimensions of the structure so it does not send more data than the allocated space allows. A socket file descriptor, a socket address length, a number of input/output vectors, a control message length and flags are sent in the initial IPC call. A length of each input/output vector is sent in a separate call afterwards. The server has all the required information at this point, so the function continues with receiving data. All parts of the message are received using IPC calls for reading large data. The socket address is received first. Data into input/output vectors are next, each in one call. Finally, the control message is received. After receiving a return value from the server, a total size of received data is retrieved as an IPC argument from the answer.

4.5 Socket server upper layer

Also called a service layer. The socket server uses the service layer to handle client connections, same as other servers. When a connection is established, it waits for client requests in an infinite loop until the client hangs up. In comparison to other network stack servers, the socket server handles all the communication in the loop. There are no callbacks to a client because all the communication is synchronous. Each IPC request is handled by one function. Possible requests are: create, bind, connect, set options, send a message, receive a message, read,

write, close and select.

Most of the functions handling IPC requests only need to retrieve arguments from the IPC call and one or two blocks of large data. Same as in the C library, the only complex parts are sending and receiving messages.

4.5.1 Sending messages

When the server is asked to send a message, it retrieves a socket file descriptor, a number of input/output vectors and flags as arguments of the IPC call. Next, a `msghdr` structure is allocated. Parts of the structure are allocated and filled as the server continues receiving data from the client. An array of vectors is allocated based on the size received as an IPC argument. The message name (socket address), input/output vectors (each separately) and control data are received as large data. All of them are received in three parts. The size of the data is first, then appropriate part of the `msghdr` is allocated, and data are received into it. At this point, the `msghdr` structure on the server is an exact copy of the one passed to the library. The socket is looked up by the file descriptor and an implementation of sending a message is looked up by the socket type. If both exist, the implementation is called with the socket, the `msghdr` structure and flags as parameters. A return value of the call is sent to the client.

4.5.2 Receiving messages

A socket file descriptor, a message name length, a number of input/output vectors, a control data length and flags are retrieved as arguments of the IPC call. A `msghdr` structure and all its parts are allocated based on the arguments. The only missing dimensions are sizes of input-output vectors. There can be many input/output vectors, so their sizes cannot be passed through IPC call arguments, even though they fit into the `sysarg_t` type. Each size is received as large data and corresponding input/output vector is allocated accordingly. A socket and an implementation of receiving a message are looked up. The implementation is called with the socket, the `msghdr` structure and flags as parameters and is expected to fill the structure with data. Contents of the `msghdr` structure are sent back to the client as large data after the call. Message name first, followed by input/ output vectors and control data. An asynchronous answer with a return value and a total size of received data is sent to the client as an IPC argument.

4.6 Socket server lower layer

The layer implements stream TCP, raw OSPF and datagram UDP sockets. It handles the communication between the upper layer and lower network stack servers. Also, UNIX sockets are emulated here.

All sockets are stored in one linked list. The list should be a sufficient data structure since there are not many socket connections expected. Many other networks stack structures (UDP associations, TCP connections, links, link addresses, etc.) are also kept in linked lists, so the overall complexity is not impaired. The list is always locked before any of the lower layer functions accesses sockets.

Each socket type is represented by a structure. At the beginning of the type-specific structure is a structure with attributes common to all sockets, making casting between sockets possible.

One of the common attributes is a socket ID. Socket IDs are generated by the socket server. The highest socket ID ever assigned and a stack of freed IDs are kept. Top of the stack is returned when there is a need for a new ID. The highest ID is incremented and returned as a new socket ID if the stack is empty. The highest ID starts at 128. The number is chosen to avoid collisions with VFS file descriptors. The highest possible VFS file descriptor is 127.

Each type of socket implements a function that returns true if there are data available for reading and false otherwise. UDP or raw socket is available for reading if its message is not empty. TCP socket is available for reading if its connection is established and there are data available on it. TCP sockets also implement a function determining if a socket is available for writing. The function returns true if the connection is established, false otherwise.

The `select` iterates through all sockets and if the socket is in the read set it calls the function determining a read availability. If the function returns true, the socket is added into the result read set. Write set is processed accordingly.

4.6.1 Raw sockets

Raw sockets are used to send and receive IP packets of higher protocols on UNIX. The higher protocol is specified by a number passed as the last argument when a socket is created. The number is compared with the protocol number in IP header when an IP packet is received. The packet is passed to all sockets matching the number. It is also passed to a higher layer of the network stack. This means that one packet can be received by multiple applications and kernel simultaneously. HelenOS network stack has limitations in this regard. They are discussed in HelenOS Chapter, Section 2.3.

Initialization

BIRD only needs raw sockets for sending and receiving OSPF packets, and there is no other server or an application that connects to the `inetsrv` in order to use the protocol.

The socket server connects to `inetrv` during the initialization, passing 89 (OSPF) as protocol number and a callback function for receiving OSPF packets. A session with `inetsrv` is created during the process.

BIRD always binds a socket to an interface right after it is created. A name of an interface is used to bind the socket, for example `net/eth1`. On UNIX, interface indexes are used to uniquely identify interfaces. When a new network interface is discovered by the `ethip` server in HelenOS, a new service for the interface is created. The service is assigned a unique ID and is added to the IP link service category. The service ID can be looked up by an interface name. A socket in HelenOS is bound to an interface by setting IP link service ID as attribute of socket structure.

Sending messages

The function for sending messages handles a parsing of the `msghrd` structure into datagram suitable for the `inetsrv` server. Datagram's IP link is given by the IP link service ID. BIRD adds always exactly one input/output vector to `msghdr` with all data. The pointer to datagram data is set to the base of the vector. The size of the data is set to the size of the vector. The destination address is retrieved from the `msghdr` socket address, and the version is set to IPv4. The source address of the datagram is more difficult to set properly because the IP link service ID is used to determine an interface. When a datagram is sent directly to IP link, none of the lower layer servers fills the source address of the IP packet generated from the datagram. In this case, source address does not matter to the sender but it is crucial to the receiver. The receiver cannot determine a packet origin without it. The solution is to set the source address to sockets IP link IPv4 address. The `inetcfg` service is used to find the IP link address. One link can have multiple addresses, so the first one returned by the `inetcfg` service is used. Datagram is passed to `inet_send` when all attributes are set.

Receiving messages

The implementation of receiving messages is split into two parts. A callback invoked by `inetsrv` and a function handling the client request.

The two parts collaborate through queues. Each socket has its own queue. The callback function iterates through all sockets whenever it is invoked. The datagram passed to the callback has assigned a link it was received from. A copy of the datagram is added to the socket queue if the socket link matches the datagram link. If there is no socket bound to the link, or there is no more memory to create a copy, the datagram is discarded.

The client request handler dequeues first message from socket's queue when called. The message is parsed into a `msghdr` structure. All raw sockets are non-blocking, so if there is no message in queue, an error code is returned. The socket address is parsed first. The source address is set to the datagram source, the port is set to zero, and the address family is set to `AF_INET`.

Actual data are parsed next. This part is a bit tricky because sending and receiving messages through raw sockets on UNIX is slightly asymmetric. Sent data does not include an IP header on UNIX. An IP header can be specified by one of the control messages optionally. Received data always contain an IP header. The `inetsrv` server is, on the other hand, symmetric and an IP header is never part of the data. Luckily, BIRD only uses the first byte of the header to determine the packet IP version and the length of the IP header. These two attributes can be reconstructed without the an actual header. They are set in the first byte. The remaining 19 bytes of the reconstructed header are nulled. The header and datagram's data are concatenated into the first input/output vector of the `msghdr` structure (BIRD always passes the `msghdr` with exactly one vector).

Additional information are last to be filled. BIRD expects one control message here. The message type is expected to be packet information and the socket option level to be IP. Data of the message is another structure with actual packet information. The structure contains two attributes - destination address of the

packet and an index of the interface, the packet was received through. The index is used to check that the packet was received through the expected interface. The destination address is used to distinguish between different interface addresses, multicast addresses and broadcast addresses. BIRD's protocols process packets differently, based on the address they were destined to. The interface index is set to the datagram IP link service ID. The destination address is retrieved from the datagram last.

4.6.2 UDP sockets

Implementation of UDP sockets is similar to RAW sockets in many ways, but there are some significant differences. The socket server obtains UDP reference during the initial connection. The reference cannot be directly used to send and receive messages. UDP associations must be created first.

Inititalization

Each UDP socket used by RIP is bound to a port and to an interface. UDP sockets are bound to links the same way as raw sockets. The UDP association is created when the `bind` is called by the client. The association takes an endpoint and a callback function as parameters. The binding is determined by the endpoint (an IP link, a local address, a local port). The IP link is set to the socket link. An address and a port are parsed from parameters of the `bind` function. One association is created for each UDP socket.

Sending messages

The `msghdr` structure must be converted into four separate parameters suitable for HelenoOS UDP API. They are a local address, a remote endpoint, data and data size. A local address is acquired in the same way as a local address for a datagram sent through a raw socket. A remote endpoint address and a port are retrieved from the `msghdr` socket address. Data are set to the base of the first input/output vector and their size to the size of the vector.

Receiving messages

The callback function and the function for receiving messages cooperate similarly as in the raw socket implementation. The callback is adding UDP messages to socket queues, and the receive function is consuming them. The difference is that UDP message can be received only by one socket. The callback does not have to iterate through all sockets to find it. It is configured during the association creation to receive the socket as a parameter.

Dequeued messages must be converted into the `msghdr` structure. The socket address is filled with a remote endpoint address and a port. The remote endpoint is a part of the dequeued message. The first input/output vector is filled with message data and its size is set to the message size.

There are no control messages expected by BIRD when receiving from UDP sockets, so the control message part is left unchanged.

4.6.3 TCP sockets

A TCP reference is obtained during the initialization. It is used to create TCP listeners and TCP connections. The socket server distinguishes between listener and non-listener TCP sockets internally.

When a listener is created, incoming connections are handled by a callback function, each in one fibril. A connection is destroyed when the function returns. The callback is adding incoming connections into the listener socket queue. The socket is determined by callback parameter. Each incoming connection is associated with a condition variable that prevents the callback function from returning. All condition variables of connections in the queue are released when a listener socket is closed.

A user can accept connections through the listener socket. A new socket is created and the first connection from the listener socket queue is assigned to it. The connection is removed from the queue. The condition variable held by the connection will be released when the newly created socket is closed. An ID of the socket and the remote endpoint of the connection are returned to the user.

A connection can be initiated by calling connect on a non-listener TCP socket. A remote address and a port are parsed from the parameter, and a new connection is created using HelenOS TCP API. The connection is added to the socket. The lifecycle of the connection is not handled by a callback function. It is destroyed when the socket is closed.

Reading and writing is trivial once the connection is established.

UNIX sockets

UNIX sockets are needed by BIRD client. They are used in the same way as TCP sockets. The only difference is that the `sockaddr` argument of bind and connect is `sockaddr_un` (unix socket path) instead of `sockaddr_in` (socket address plus port). UNIX sockets are emulated with localhost TCP sockets. Each UNIX socket has a TCP socket assigned to it on the server side. When any of the functions (bind, listen, connect, accept, write, read) is called on UNIX socket, the call is delegated to the TCP socket. Additionally, bind assigns a local port to the path (for example, 9000 is assigned to "bird.ctl") and passes it to TCP socket bind along with localhost address (127.0.0.1). Listen adds the path-port pair into the list of paths that is being listened on. The connect looks up the port in the list according to the path it wants to connect to. If found, it binds it self to another port and calls connect on the TCP socket. Accept creates a new UNIX and assigns it the TCP socket from the TCP accept call.

Here is an example of a server and a client communicating over "bird.ctl" path. There is a TCP socket listening on 127.0.0.1:9000 mapped to "bird.ctl" path by UNIX socket. The client uses a UNIX socket with TCP socket bound to 127.0.0.1:9001. The server uses another UNIX socket obtained from accept with a TCP socket bound to 127.0.0.1:9002.

4.7 Additional changes

The Section describes an implementation of requirements from 3.5. It was tried to implement the requirements in a least invasive way possible.

4.7.1 Multicast

A driver service ID is added to the network layer link information. BIRD can now use the driver API to enable or disable the multicast on an interface given by the ID. Link information is accessed through the `inetcfg` service of the `inetsrv` server. The server acquires a driver service ID from the `ethip` server in a separate IPC call during a link opening (same as other information like an MTU, a MAC address, etc.).

Translations of two multicast addresses are added to `ethip` ARP protocol. They are OSPF and RIP multicast addresses, specified by RFC [12] and [10], respectively. Both are translated to Ethernet addresses according to rules of multicast address translation, described in RFC [4], Section 6.4. Additionally, network layer was made aware of them. Server `inetsrv` now treats packets destined to these two multicast addresses as packets destined to one of the local addresses.

4.7.2 Deleted interface addresses and routes, route origin

A list of deleted interface addresses is added. Interface addresses are moved from the list of active addresses into the list of deleted addresses instead of being deleted. A new interface address is added to the list of active addresses. Additionally, it is looked up in the list of deleted addresses. If a match is found, the old deleted address is removed and deallocated. Matching criteria include all attributes except for the name.

The library API for listing routes and interface addresses is changed as well. The API now takes an interface address status as a parameter. The status can be active or deleted. When IPC request arrives to the server, implementation will return addresses from the appropriate list.

Deleted routes and the route origin are described in the Section 4.7.4.

4.7.3 Routing

The integrated implementation of the routing from the thesis Packet filtering [3] takes advantage of the fact that HelenOS packets with local origin were already routed. The `inetsrv` is deciding if a packet is destined to one of the local addresses. If not, it is routed the same way as packets with local origin instead of being discarded.

There was one problem with the implementation. Packets were forwarded only if their destination address matched network address of one of the interfaces. It caused HelenOS to route only one hop. Removal of the unnecessary check fixed the issue.

4.7.4 Routing table data structure

A binary trie data structure with no further modifications is added to the C library. Implemented operations are insert, find longest match and find exact. Delete is not implemented because routes are never really deleted, only marked as such. The data can be inserted and looked up under keys with different bit lengths.

The `intesrv` server uses two separate tries as routing tables for IPv4 and IPv6 routes. The destination address type of a route is deciding which trie is used. A path in the trie corresponds to a destination address. The path ends with a node containing a pointer to a list of routes (there can be multiple paths to one destination).

Following steps are executed when a new route is created. A path in the trie is found according to the destination address and prefix. If the node at the end of the path already contains a list of routes, the new route is prepended to the list. Otherwise, a new list is created, the route is inserted into it and the list is inserted into the trie.

A route is looked up by a destination address, a prefix length and a router address when it is being deleted. A list is looked up by a destination address and a prefix in the trie. If the list is found, it is searched for a route matching the router address. The route is moved from its current position to the end of the list and its status is changed to inactive.

A packet destination address may match multiple entries in a routing table. For example, address 1.1.1.1 matches both 1.0.0.0/8 and 1.1.1.0/24. The longest prefix match must be chosen. A trie function is implemented for this purpose. A list of routes with longest destination address prefix match is looked up when a packet needs to be routed. The first route from the list is used, if it is active. The maximum depth of the trie is 32 for IPv4 addresses. The complexity of searching longest prefix routes is not impaired by the lists. Only deleting and adding new routes routes can have increased complexity (the route is additionally searched in the list), and that is only if there are multiple routes to the same destination.

Routes are allocated in blocks, since they are newer really deleted. One block has space for 1024 routes. A new block is allocated whenever the space is depleted. The blocks are kept in a list. It is used for listing all routes (there is no need to convert the trie into an array or a list). The size of a block is slightly smaller than the maximum IPC transfer size. Each block from the list is sent to a client in a separate IPC call for transferring large data.

Two route attributes were removed during the routing table data structure reimplementation - a route ID and a route name. It does not make much sense to look up routes by any of the two attributes when routes are stored in a trie under destination addresses. An attribute determining route origin was added. It is up to a user of the library to fill the attribute properly during route creation.

4.7.5 Source address of sent UDP messages

A parameter specifying source address was added to function for sending messages through UDP association in the C library. It is passed to the server with other parameters. The server handles the source address parameter similarly to the remote endpoint parameter. If the passed local address is null, the local address

specified by the association local endpoint is used (it can still be null, in which case the source address of the message will be also null). Otherwise, the passed address is used as the source address of the message.

The link can be the only endpoint pair attribute specified during the association creation. The source address can be supplied with each message. The case, in which the link address changes, is also covered.

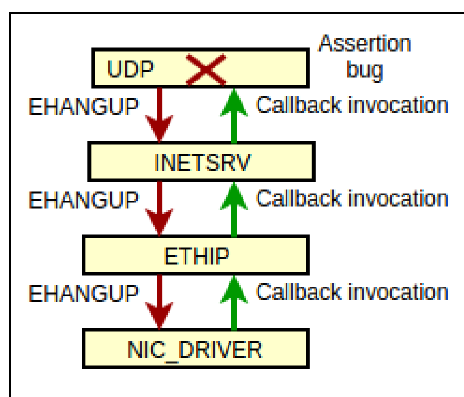
4.7.6 Network stack bug fix

The following text discusses the fix of the bug described in the analysis Chapter, Section 3.5.3. The bug did not occur when the latest official stable release of HelenOS was used instead of the image compiled from the latest repository version. This meant the bug was introduced in one of the revisions between the latest release and the current revision. It turned out to be the revision, where the transport layer was redesigned (sockets were replaced with TCP and UDP servers). The conclusion was that the bug was caused by one of the servers.

There was found a typo error in an assert in the UDP server. When the server received UDP packet, it tried to look up a destination association. The association is most commonly looked up by a port. When the association was not found, the server crashed. The correct behavior is to discard the packet.

The bug was discovered because the other network participants were broadcasting some UDP packets and there was no consumer for them in HelenOS. It was easily removed by fixing the typo in the assert. The implication of the bug is that a failure in one of the high layer servers caused a harm to the whole network stack. A possible cause is error propagation in the network stack. The UDP server crash led to the `EHANGUP` error code propagation as a return value of each callback invocation. The `EHANGUP` was returned all the way down to the driver as illustrated by the Figure 4.2.

Figure 4.2: Hangup propagation.



The driver failed to invoke callback of the `ethip` for all subsequently received messages. The network stack remained functional only for sending.

4.8 BIRD's HelenOS system-dependent layer

The layer is implemented in HelenOS coastline and is added to BIRD during the coastline build process. The `inetcfg` library is used to implement the synchronization with HelenOS.

4.8.1 Interface scanning

IDs of all links are listed first. A link information is retrieved for each link ID. Following attributes are extracted from the link information: an interface name, a link service ID and an MTU. Link service ID is used as an interface index. Temporary BIRD interface structure with these attributes is created and passed to the core. Active and deleted interface addresses are acquired next. Following attributes are extracted from each interface address information: an interface address, an interface address prefix length and a link service ID as an interface index. Other attributes (e.g. a scope, a broadcast address, a network address) are calculated next. A core address update is called for each active address. Deleted addresses are deleted from the core.

4.8.2 Routing table scanning

The HelenOS API for getting routes was rewritten. All the routes are loaded in one call. A destination address, a destination address prefix length, a router (gateway) address and a route origin are acquired from each route information. On Linux, deleted routes are handled by reading the RTNETLINK kernel socket and invoking a hook as described in the Section 1.5. This is substituted in HelenOS by storing deleted routes and processing them during the route scan. Deleted routes, unlike deleted interface addresses, are not stored on Linux.

4.8.3 Creating and deleting routes

A BIRD internal structure for a routing table entry is passed to the function for creating or deleting routes. A destination address, a destination address prefix length and a router (gateway) address of the structure are used. A route matching these attributes is looked up in HelenOS routing table. If a match is not found and the route is being created, a new route with these attributes is added to the routing table. A matching route is looked up to avoid duplicates. If a route is being deleted and a match is found, the route is deleted from the HelenOS routing table.

4.8.4 BIRD client

On UNIX, the client's main loop is blocked by select until either an input is available or data from the server arrived. The loop is redesigned as follows. It is blocked by waiting for input (fgets). When the input is typed in, waiting for response from server starts. After the response is processed, waiting for another input starts. There is an issue with displaying the typed input, which is printed only once submitted. The HelenOS console does not display the input when another application is running in it, so the commands must be typed in blindly.

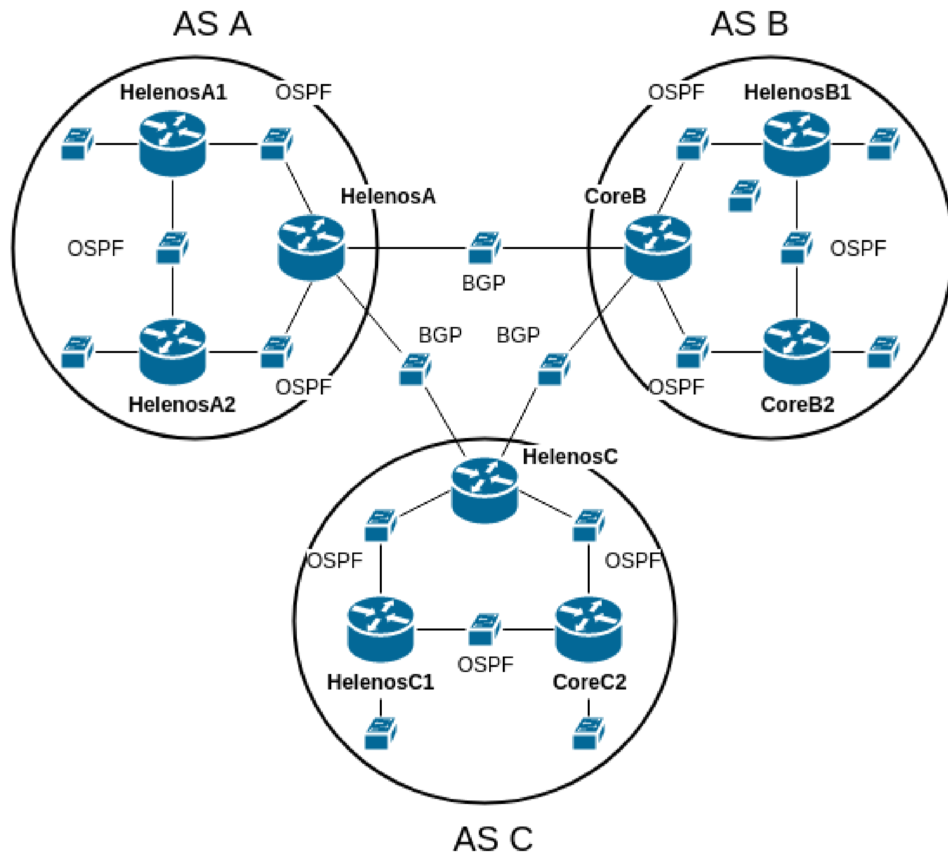
5. Evaluation

The following Chapter describes evaluation of the implementation in a virtual environment. The same environment was used to perform all the tests except.

5.1 Environment setup

The implementation was tested in the virtual environment proposed in 3.1. The Appendix B contains description how to run the testing environment. The topology consists of nine routers. Six of them are instances of HelenOS and three of them are instances of Core UNIX. A script is prepared for each instance to configure the network and run BIRD. The configuration results in a network displayed by Figures 5.1, C.1, C.2 and C.3. The Figure 5.1 shows a high level view. A more detailed view of autonomous systems is displayed in the Appendix C by Figures C.1, C.2 and C.3.

Figure 5.1: Overall topology



5.1.1 BIRD configuration

The configuration file is passed as an argument when BIRD is started by one of the scripts. The nodes connecting "AS"s (HelenOS_A, Core_B and HelenOS_C) have configured BGP on interfaces between them. There are two instances of the

BGP protocol configured on each of the nodes. The BGP instances are configured to import and export all routes. OSPF is configured on the interfaces connected to inner nodes.

The inner nodes (HelenOS_A1, HelenOS_A2, HelenOS_B1, Core_B2, HelenOS_C1, Core_C2) have configured OSPF for all interfaces. OSPF configuration is the same for all interfaces on all nodes. The protocol imports and exports all routes. The connection type is set to point to point since each interface is connected to none or exactly one other router. The period for sending hello packets is set to fifteen seconds, retransmission after not getting an acknowledgement to ten seconds, waiting after the start for an adjacency build to twenty-five seconds and the number of seconds before a neighbor is proclaimed dead to forty.

The kernel and the device protocols are also configured in a same way on all nodes. The routing table is scanned every five minutes. Interfaces are scanned every five seconds.

5.2 Tests

The initial exchange of routing information happens when the network is configured and BIRDs are started. The routing information is collected by OSPF based on configured network interface addresses within each of the three "AS"s. BGP exchanges the information between the "AS"s.

Each AS contains five destination addresses (they are 10.10.10.0/24, 10.10.20.0/24, 10.10.30.0/24, 10.10.40.0/24, 10.10.50.0/24 in AS A). This means nodes connecting "AS"s should have installed thirteen routes by BIRD after the initial exchange is over (fifteen destinations minus two destinations given by the interface addresses). The inner nodes should have installed twelve routes each (three destinations are given by interface addresses). Tables 5.1 and 5.2 show how should the routing tables look like on HelenOS_A and HelenOS_A1 respectively (routes can be displayed in arbitrary order).

Table 5.1: HelenOS_A routing table after initial exhange.

Route index	Destination network	Gateway(Router)
1.	10.10.30.0/24	10.10.10.2
2.	10.10.40.0/24	10.10.10.2
3.	10.10.50.0/24	10.10.20.2
4.	20.10.10.0/24	1.0.0.2
5.	20.10.20.0/24	1.0.0.2
6.	20.10.30.0/24	1.0.0.2
7.	20.10.40.0/24	1.0.0.2
8.	20.10.50.0/24	1.0.0.2
9.	30.10.10.0/24	2.0.0.2
10.	30.10.20.0/24	2.0.0.2
11.	30.10.30.0/24	2.0.0.2
12.	30.10.40.0/24	2.0.0.2
13.	30.10.50.0/24	2.0.0.2

Table 5.2: HelenOS_A1 routing table after initial exhange.

Route index	Destination network	Gateway(Router)
1.	10.10.20.0/24	10.10.30.2
2.	10.10.50.0/24	10.10.30.2
3.	20.10.10.0/24	10.10.10.1
4.	20.10.20.0/24	10.10.10.1
5.	20.10.30.0/24	10.10.10.1
6.	20.10.40.0/24	10.10.10.1
7.	20.10.50.0/24	10.10.10.1
8.	30.10.10.0/24	10.10.10.1
9.	30.10.20.0/24	10.10.10.1
10.	30.10.30.0/24	10.10.10.1
11.	30.10.40.0/24	10.10.10.1
12.	30.10.50.0/24	10.10.10.1

5.2.1 OSPF test

The OSPF test is performed in the AS A. Network changes are simulated in this test by removing and adding interface addresses on involved nodes. Changes in routing tables are observed to confirm that they match the network state.

When a change is made to the network, some period of time needs to elapse before BIRDs exchange all the information and update routing tables. Waiting one minute after each change should be enough for all updates to take place.

All the tests are interactive. After modifying network configuration, and waiting for approximately one minute, results are verified by examining routing tables on all routers. Additionally, destinations of changed routes are pinged from some of the nodes. BIRD clients are used to show state of the OSPF protocol during the tests.

Labels are introduced on destination addresses and network interface addresses to simplify the description of the tests.

Table 5.3: Destination network address labels

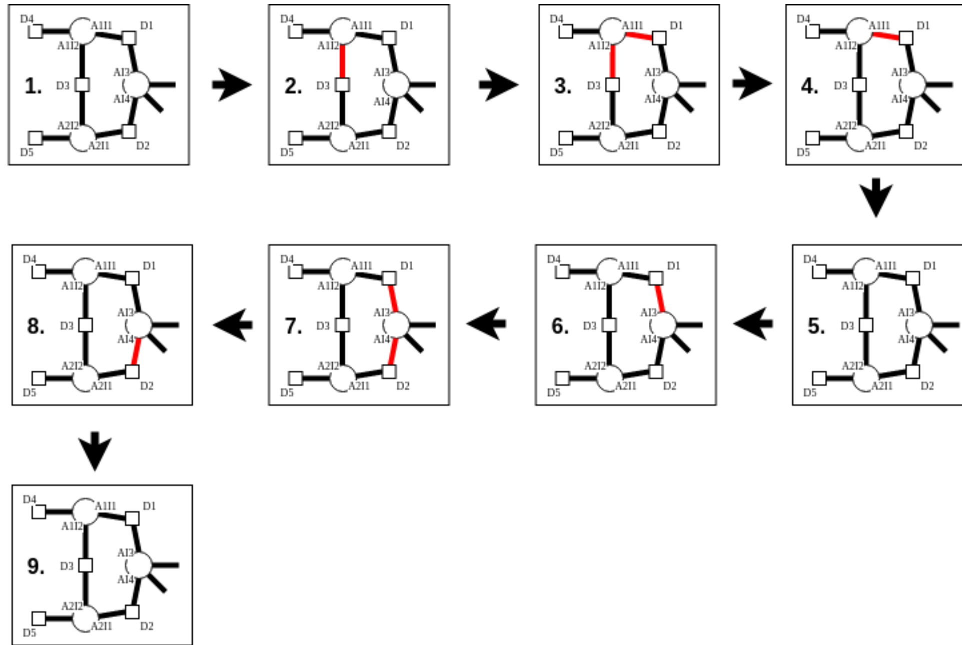
Label	Description
D1	network between HelenOS_A and HelenOS_A1, 10.10.10.0/24
D2	network between HelenOS_A and HelenOS_A2, 10.10.20.0/24
D3	network between HelenOS_A1 and HelenOS_A2, 10.10.30.0/24
D4	network behind HelenOS_A1, 10.10.40.0/24
D5	network behind HelenOS_A2, 10.10.50.0/24

Table 5.4: Network interfaces address labels

Label	Description
AI3	address of net/eth3 interface on HelenOS_A, 10.10.10.1/24
AI4	address of net/eth4 interface on HelenOS_A, 10.10.20.1/24
A1I1	address of net/eth1 interface on HelenOS_A1, 10.10.10.2/24
A1I2	address of net/eth2 interface on HelenOS_A1, 10.10.30.1/24
A2I1	address of net/eth1 interface on HelenOS_A2, 10.10.20.2/24
A2I2	address of net/eth2 interface on HelenOS_A2, 10.10.30.2/24

The Figure 5.2 shows network states in the AS A during the test. States are numbered one to nine. Changes between states 2-3-4 and 6-7-8 affect all "AS"s. Changes between other states affect only the AS A.

Figure 5.2: OSPF Test states



Description of test steps follows.

- A1I2 is deleted in the first step (state 2). HelenOS_A changes the gateway of the route into the D3 from A1I1 to A2I1, if it was A1I1 previously (distance is the same through both of them). HelenOS_A2 changes the gateway of routes into D4 and D1 from A1I2 to AI4. HelenOS_A1 changes the gateway of routes into D5 and D2 from A2I2 to AI3. Additionally, HelenOS_A1 adds a new route into D3 with AI3 as a gateway. The route was previously given by the deleted interface address, now it is accessible only through HelenOS_A. Other autonomous systems are unaffected by this step.

- The second step is to delete A1I1 (state 3). HelenOS_A1 becomes separated from the rest of the network. All routers in all "AS"s remove the route into D4. HelenOS_A1 deletes all routes from its routing table.

- The A1I2 is restored next (state 4). HelonOS_A1 becomes part of the network again. The previously removed route is reinstalled by all nodes and HelenOS_A1 reinstalls routes to all twelve destinations.

- The network returns to initial state after A1I1 is restored (state 5). One route needs to be shortened on both HelenOS_A1 and HelenOS_A2. HelenOS_A1 changes the gateway of the route into D5 from AI3 to A2I2. HelenOS_A2 changes the gateway of route into D4 from AI4 to A1I2. HelenOS_A1 deletes route into D1 (the route is now given by A1I1 again).

- Deletion of AI3 follows (state 6). HelenOS_A2 changes the gateway of the route into the D1 from AI4 to A1I2, if it was AI4 previously. HelenOS_A1

changes the gateway of route into D2 from AI3 to A2I2 if necessary. HelenOS_A changes the gateway of routes into D4 and D3 from A1I1 to A2I1. Additionally, HelenOS_A adds a new route into D1 with A2I1 as a gateway.

- HelenOS_A1 and HelenOS_A2 are separated from the rest of the network by deleting AI4 (state 7). All the other nodes delete routes to all AS A destinations (D1, D2, D3, D4, D5). HelenOS_A1 and HelenOS_A2 delete all routes into other "AS"s but they maintain the routes within AS A.
- AI3 is restored and HelenOS_A1 and HelenOS_A2 are reconnected to the network (state 8). Each of them reinstall ten routes into other "AS"s. Nodes in AS B and AS C reinstall the five routes into AS A. HelenOS_A adds routes into D2, D3, D4 and D5. The gateway of the routes is A1I1.
- The network returns to initial state again when AI4 is restored. HelenOS_A deletes route into A1 and changes router of route into D5 from A1I1 to A2I2.

5.2.2 BGP test

The BGP test follows after the OSPF test is concluded. Bird on Core_B node is killed. Routes into the AS B are deleted on nodes in the AS A and the AS C. Remaining nodes in the AS B (HelenOS_B1 and Core_B2) delete routes into the other two "AS"s. Bird on Core_B is restarted with the same configuration on Core_B after all exchanges are over. The routes are reinstalled on all nodes and the network is its initial state.

5.2.3 Routing performance test

A configuration is changed on the Core_B2 node from bird.conf to birdB2.conf using the BIRD client. The static protocol in the birdB2.conf configuration file contains ten thousand randomly generated routes. The routes are spread to all the nodes. The nodes connecting autonomous systems have 10013 routes and the inner nodes have 10012 routes added by BIRD. A new command, `log-sr` is implemented for the `inet` tool in HelenOS. It prints routes into the file `routes.log` and prints their count to the console.

An application called `packet-generator` is prepared on Core_C2 node. The application has a hard-coded array of ten thousand destinations. Each destination corresponds to one of the ten thousand randomly generated routes. The application generates and sends one thousand packets. A random destination from the array is picked for each packet. Source code of the application can be found on CoreB2 image.

It took HelenOS approximately twenty seconds to route one thousand packets in the current testing environment. Incoming packets are observed on Core_B node with tcpdump. The time is calculated as a difference between timestamps of the first and the last packet.

The routing table data structure is not a bottleneck for the performance. The same time is needed to route the packets even if the destination is hard-coded and the routing table is not even consulted.

5.2.4 RIP test

The RIP protocol was tested in a less complex environment. Two issues were found with this protocol. When routes are deleted on one router, other routers do not react to this change. When the protocol was tested between three routers, one of them did not choose the shortest possible routes. Both of these problems are not HelenOS related because they occurred even when the HelenOS was swapped for a UNIX Core. The RIP protocol was redesigned in a more recent version of BIRD, so the issues were not reported. The RIP protocol is not included in the final testing set for these reasons.

Conclusion

Goals of the thesis were partially achieved. HelenOS provides support for BIRD's three main routing protocols - OSPF, RIP and BGP. HelenOS can be used as a routing OS in an IPv4 environment.

Network sockets were implemented in HelenOS in a scope necessary for the three protocols. Sockets were implemented in a new server. The socket API was added to the C and the POSIX libraries. The design respects the HelenOS micro-kernel multiserver architecture. Even though the created socket implementation only includes a functionality needed by BIRD, it is easily extensible and ready to be used by other POSIX and HelenOS native applications.

Several additional changes were made to the HelenOS network stack to make BIRD functional. A new HelenOS lower system-dependent layer was implemented in BIRD. HelenOS routing table data structure was upgraded to a trie. The trie implementation is also reusable.

A virtual network environment was created for testing. Multiple changes were made to the network configuration while BIRDs were running. All changes were correctly communicated between all nodes in case of OSPF and BGP protocols. The RIP protocol was functional only in a static environment. Tests prove that BIRD's HelenOS layer and sockets are functional. The environment is also suitable for testing other HelenOS networking related features.

Future work

There are BIRD functionalities which are still not supported by HelenOS.

To support IPv6, HelenOS network stack needs multiple IPv6 related upgrades. Most of them are discussed in thesis IPv6 for HelenOS [13].

Part of another thesis, Port of QEMU to HelenOS [11], dealt with implementation of POSIX threads. Unfortunately, the implementation is not merged into the HelenOS trunk. If a merge of this branch into the trunk happens in the future, it should be simple to support the BFD protocol.

New versions of BIRD with new features are released continually. The ported version can be upgraded, when requirement on some of the latest features arises. This thesis should have laid enough ground work to make the upgrades relatively simple.

There are still many unmet RFC requirements on HelenOS both as a host and as a router that needs to be implemented to achieve compliance.

Bibliography

[1] Protocol numbers. URL https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml. (as of May 1st 2018).

[2] Helenos, January 2015. URL http://www.helenos.org/. (as of May 1st 2018).

[3] Jan Buchar. Helenos packet filter, 2015. URL http://www.helenos.org/doc/theses/jb-thesis.pdf. (as of May 1st 2018).

[4] S. Deering. OSPF Version 2. RFC 1112, RFC Editor, August 1989. URL https://tools.ietf.org/html/rfc1247. (as of May 1st 2018).

[5] Editor F. Baker. Requirements for IP Version 4 Routers. RFC 1716, RFC Editor, June 1995. URL https://tools.ietf.org/html/rfc1812. (as of May 1st 2018).

[6] Ondřej Filip, Pavel Machek, Martin Mareš, and Ondřej Zajíček. Bird programmer's documentation, April 2015. URL ftp://bird.network.cz/pub/bird/1.5/bird-doc-1.5.0.tar.gz. (as of May 1st 2018).

[7] Ondřej Filip, Pavel Machek, Martin Mareš, and Ondřej Zajíček. Bird user's guide, April 2015. URL ftp://bird.network.cz/pub/bird/1.5/bird-doc-1.5.0.tar.gz. (as of May 1st 2018).

[8] Stanislav Gálfy. Helenos ticket 667, ping not working, May 2016. URL http://www.helenos.org/ticket/667#. (as of May 1st 2018).

[9] Stanislav Gálfy. Helenos ticket 672, udp crashing network stack, February 2017. URL http://www.helenos.org/ticket/672. (as of May 1st 2018).

[10] G. Malkin. RIP Version 2. RFC 1654, RFC Editor, November 1998. URL https://tools.ietf.org/html/rfc2453. (as of May 1st 2018).

[11] Jan Mareš. Port of qemu to helenos, 2015. URL http://www.helenos.org/doc/theses/jm-thesis.pdf. (as of May 1st 2018).

[12] J. Moy. OSPF Version 2. RFC 1247, RFC Editor, July 1991. URL https://tools.ietf.org/html/rfc1247. (as of May 1st 2018).

[13] Antonín Steinhauser. Ipv6 for helenos, 2013. URL http://www.helenos.org/doc/theses/jb-thesis.pdf. (as of May 1st 2018).

[14] Wikipedia. Border gateway protocol, September 2004. URL https://en.wikipedia.org/wiki/Border_Gateway_Protocol. (as of May 1st 2018).

[15] Wikipedia. Open shortest path first, June 2005. URL https://en.wikipedia.org/wiki/Open_Shortest_Path_First. (as of May 1st 2018).

[16] Wikipedia. Routing information protocol, September 2005. URL https://en.wikipedia.org/wiki/Routing_Information_Protocol. (as of May 1st 2018).

List of Abbreviations

API - Application Programming Interface
AS - Autonomous System
BFD - Bidirectional Forwarding Detection
BGP - Border Gateway Protocol
BIRD - BIRD Internet Routing Daemon
BSD - Berkeley Software Distribution
CPU - Central Processing Unit
GNS3 - Graphical Network Simulator-3
ICMP - Internet Control Message Protocol
IP - Internet Protocol
IPC - Inter Process Communication
IPv4 - Internet Protocol version 4
IPv6 - Internet Protocol version 6
MAC - Media Access Control
MD5 - Message Digest 5 Algorithm
MTU - Maximum Transmission Unit
NIC - Network Interface Controller
OS - Operating System
OSPF - Open Shortest Path First
PDU - Protocol Data Unit
POSIX - Portable Operating System Interface
RADV - Router Advertisement Daemon
RIP - Routing Information Protocol
TCP - Transmission Control Protocol
UDP - User Datagram Protocol
VFS - Virtual File System
VM - Virtual Machine

Appendices

A. Electronic attachment

Content of archive uploaded in Student Information System as electronic attachment of the thesis.

- HelenOS/mainline - clone of repository
  `https://github.com/StanislavGalfy/helenos.git`
  branch `bird-port-unmerged`
  - bird port mainline implementation, merged with master on 19.3.2018 (head 973be38). Branch `bird-port` was merged later, but merge was reverted into this branch because of coastline build errors.

- HelenOS/mainline/image.iso - pre-built image of HelenOS.

- HelenOS/coastline - clone of repository
  `https://github.com/StanislavGalfy/harbours.git`
  branch `bird-port`.
  - bird port coastline implementation, merged with master head 453d818.

- HelenOS/topology - clone of repository
  `https://github.com/StanislavGalfy/topology.git`
  branch `master`
  - Contains testing environment images and scripts.

- Diffs/mainline-status-diff.txt - list of mainline modified/added files.

- Diffs/coastline-status-diff.txt - list of coastline modified/added files.

- Diffs/mainline-diff.txt - full mainline diff, can be also seen in pull request on github (https://github.com/StanislavGalfy/helenos/pull/2/files).

- Diffs/coastline-diff.txt - full coastline diff, can be also seen in pull request on github (https://github.com/StanislavGalfy/harbours/pull/1/files).

B. Compiling and running

Prerequisites:

- VDE2
- QEMU - compiled with `--enable-vde` option. QEMU is installed without enabled VDE when package manager is used in some cases. Install QEMU with script included in HelenOS mainline `contrib` folder to ensure this option is enabled.
- GIT - optional, for getting sources from online repositories

All source codes can be found in the electronic version of the thesis or in online repositories as shown below.

To download repositories, git must be installed. Rest of the text will assume, all three are cloned into `~/HelenOS/` folder.

```
mkdir ~/HelenOS
git clone https://github.com/StanislavGalfy/helenos.git
    ~/HelenOS/mainline
git checkout bird-port-unmerged
git clone https://github.com/StanislavGalfy/harbours.git
    ~/HelenOS/coastline
git checkout bird-port
git clone https://github.com/StanislavGalfy/topology.git
    ~/HelenOS/topology
```

If sources from the electronic version are used, it will be assumed, that the `HelenOS` folder is copied into the home folder. The electronic version also contains a pre-built image of HelenOS, so steps up to the start of the virtual environment can be skipped.

The next step is to initialize a directory for coastline build, as instructed in the coastline readme.

```
cd ~/HelenOS/build-ia32
~/HelenOS/coastline/hsct.sh init  /HelenOS/mainline ia32 build
```

At this point, BIRD is ready to be installed into HelenOS.

```
~/HelenOS/coastline/hsct.sh install bird
cd ~/HelenOS/mainline
make
```

The virtual environment can be started now.

```
cd ~/HelenOS/topology
./topology.sh
```

The script uses the image of HelenOS created with previous steps for six of the nine nodes.

Another script, `./run-birds.sh`, is provided to start birds on all nodes. The script uses QEMU monitor, so it should be used only once all nodes are fully booted. The script runs all node-specific scripts for configuring network and starting BIRD.

C. Detailed configuration of autonomous systems

Figure C.1: Network A topology

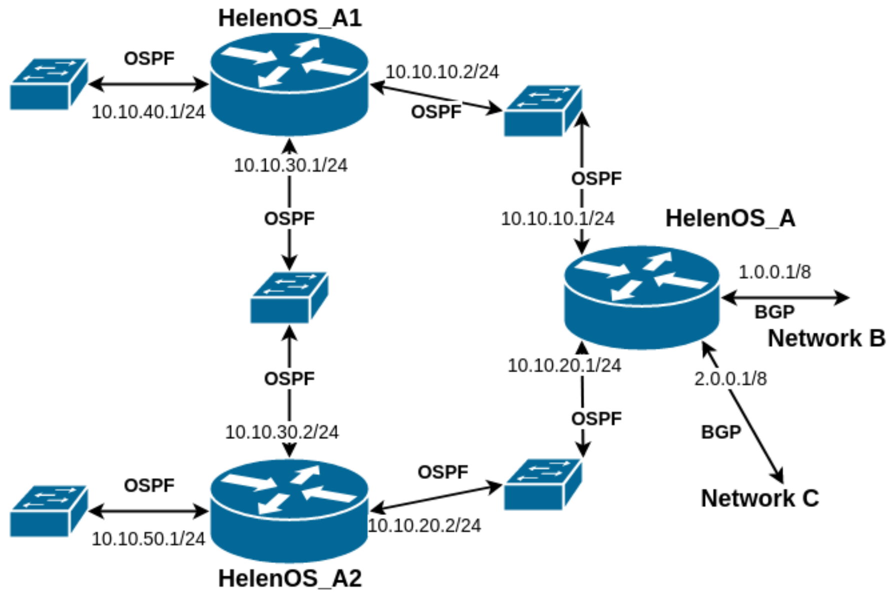


Figure C.2: Network B topology

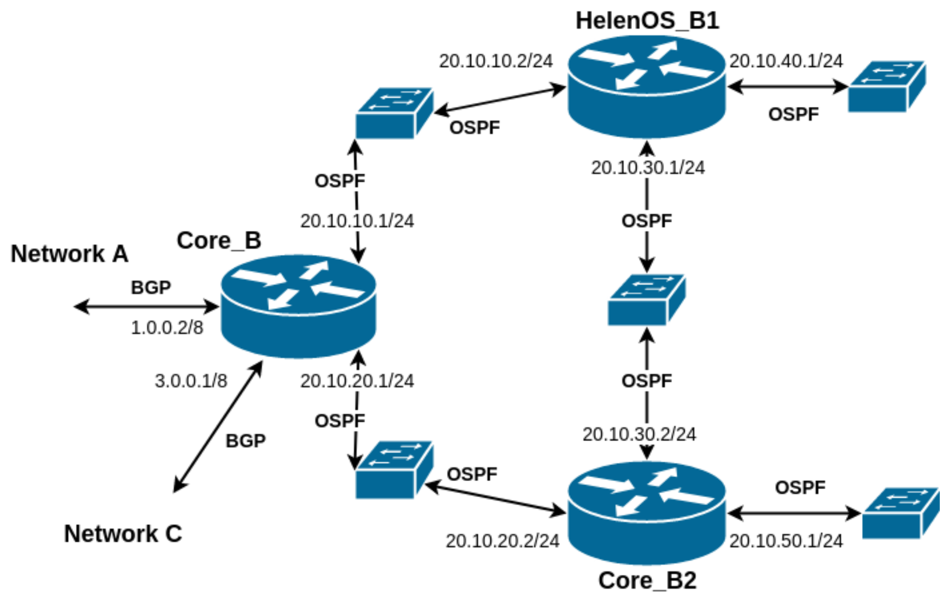


Figure C.3: Network C topology