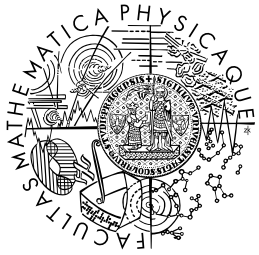


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 2017

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 ..... date ..... 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 one of key features of modern multipurpose operating system, which HelenOS aims to be. Goal of this master thesis is to explore current HelenOS routing capabilities, enhance them and empower HelenOS with BIRD. Thanks to BIRD, HelenOS will become a routing operating system, that is aware of its surroundings and is capable of dynamic adaptation to changes in network, it is part of.

Keywords: HelenOS, BIRD, networking, routing, microkernel

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

Contents

Introduction

Motivation and goals

This thesis aims to describe upgrade of HelenOS from 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 for academic purposes. 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 creation of environment where the implementation will be tested.

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

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

Content

The first chapter describes BIRD on a UNIX-like systems with emphasize on system dependent parts.

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

Analysis can be found in third chapter. It describes prerequisites, possible approaches to porting BIRD and port requirements on network stack and BIRD.

The fourth chapter is dedicated to implementation of libraries required to port BIRD, changes in network stack and changes in BIRD.

The fifth chapter evaluates the whole work by describing a series of tests in 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. Practically it is used on some of the important Internet nodes, for example in Moscow, Milan, London.

BIRD's main role in routing process is to keep 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 operating system to achieve this goal are network interface scanning, routing table scanning, writing entries into routing table, sending and receiving TCP, UDP and IP messages.

Following text first briefly describes BIRD's overall structure, execution flow, then focuses on OS dependent parts. At the end of the chapter there are briefly described relevant protocol implementations.

1.1 Version

A new version of BIRD is released approximately every three to four months. Version ported to HelenOS is 1.5.0, which was the actual version of BIRD when work on this theses started. The whole text of the theses refers to this version. Documentation corresponding to it can be found archived at [11] and [12].

1.2 BIRD modules

BIRD functionality is split into seven main modules. Core of the BIRD is implemented in module called `nest`. It contains structures and functions for storing and handling all data collected by BIRD. They can be collected during synchronization with OS (e.g. network interface info, local OS routes) or by one of the routing protocols (e.g. routes advertised by other routers, neighbor info).

Module `sysdep` contains a system dependent code. Separate section 1.5 is dedicated to description of this module.

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

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

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

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

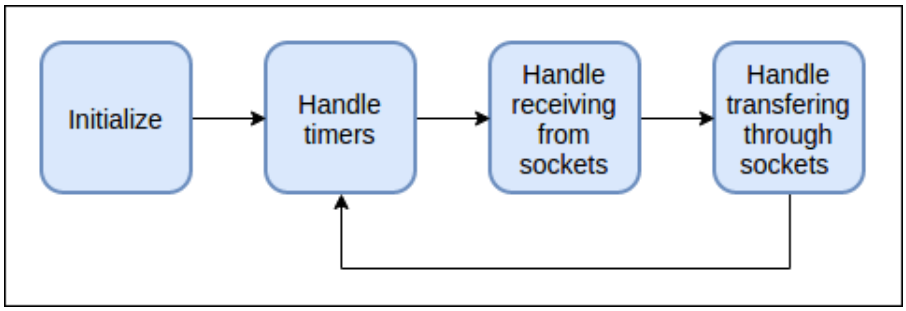
The last to mention is `client` module. BIRD client, which is a separate binary file, is compiled from this module's code. Client can be used for communication with running BIRD. It can display info about BIRD status, running protocols,

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, with main loop. There is one exception, covered in section 1.4 . A figure 1.1 shows key points of application execution.

Figure 1.1: Bird execution flow.



Initialization

During this phase modules and resources are initialized, test for other running instance of BIRD is performed, user ID and group ID are set, process is daemonized by forking. The most important part of this phase is reading and parsing configuration for the first time. According to configuration, protocols are initialized. During protocol initialization sockets are created and configured, timers for protocol specific events are created, receive and transfer hooks are configured.

Timer handling

When the initialization phase is over, BIRD enters main loop. At the beginning of the loop, time is updated. Next, the timers are examined. Hook functions are executed on the ones that expired. This is the point where following operations are realized. Synchronization with OS routing table and interfaces, protocol message transfer through sockets, rereading of configuration file, etc. Most of the timers are recurring, so after executing the hook they are rescheduled.

Socket reading

After timers are handled, BIRD executes select on two sets of sockets. First set contains all known sockets with associated receive hooks, second with transfer hooks. One socket can belong to both sets. After the select, first set contains sockets that are available for reading without blocking. Second set contains sockets that are ready to transfer.

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

If there are data available for reading on a socket, they are retrieved and passed to receive hook associated with socket. Receive hook processes the data

according to protocol and retrieved information is propagated to the core and possibly to other running protocols.

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

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

Socket writing

Protocols can either send messages in predefined intervals (in which case timers are used) or send messages whenever socket is available (in which case they are sent at this point). Set of sockets ready to transfer is determined in the same select as set of sockets with data available for reading. If socket is ready to transfer, transfer hook associated with it is called. More common approach to sending protocol messages is to use timers.

1.4 Threads

BIRD documentation on this topic is slightly inconsistent. Both in design goals and task documentation is mentioned that BIRD is a single threaded application, but BFD protocol documentation states this protocol is partially running in a separate thread.

"Design goals.

Respond to all events in real time. A typical solution to this problem is to use lots of threads to separate the workings of all the routing protocols and also of the user interface parts and to hope that the scheduler will assign time to them in a fair enough manner. This is surely a good solution, but we have resisted the temptation and preferred to avoid the overhead of threading and the large number of locks involved and preferred a event driven architecture with our own scheduling of events. An unpleasant consequence of such an approach is that long lasting tasks must be split to more parts linked by special events or timers to make the CPU available for other tasks as well.[11]"

"Since BIRD is single-threaded, it requires long lasting tasks to be split to smaller parts, so that no module can monopolize the CPU.[11]"

"The BFD implementation uses a separate thread with an internal event loop for handling the protocol logic, which requires high-res and low-latency timing, so it is not affected by the rest of BIRD, which has several low-granularity hooks in the main loop, uses second-based timers and cannot offer good latency. The core of BFD protocol (the code related to BFD sessions, interfaces and packets) runs in the BFD thread, while the rest (the code related to BFD requests, BFD neighbors and the protocol glue) runs in the main thread.[11]"

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

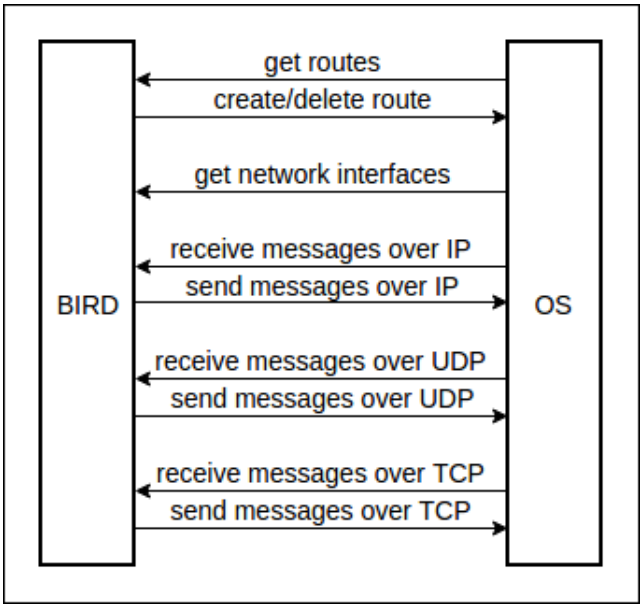
1.5 System dependent parts

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

- upper layer, called `unix`, is code common for Unix-like systems
- lower layer is OS family specific code. Implementation is chosen based on target OS, configured before compiling. There are two implementations of lower layer.
  - `linux` - linux OS family specific code.
  - `BSD` - BSD OS family specific code.

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

Figure 1.2: Bird OS communication.



1.5.1 Unix

This layer implements main function and main loop described in section 1.3. Moreover there are implemented functions for handling sockets, files, timers and synchronization with OS.

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 already running instance of BIRD, and also for communication with BIRD client.

Code dealing with OS synchronization is implemented by two protocols, `proto_unix_kernel` and `proto_unix_iface`. First for routing table synchronization and second for interface synchronization. They are represented by the same structures as routing protocols, defining hooks for events like protocol start, reconfiguration, preconfiguration, postconfiguration, shutdown etc.

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

Time updates can be handled in two ways. If OS provides monotonic clock implementation, it will be used. Otherwise they are handled by BIRD internal clock implementation.

1.5.2 Linux

Linux layer implements synchronization with OS through kernel sockets. These sockets have `PF_NETLINK` domain, `SOCK_RAW` type and `NETLINK_ROUTE` protocol. They are called rtnetlink sockets. 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 asynchronous communication with OS.

Next are described four most important functions implemented here, that upper layer uses to handle synchronization with OS.

- `void kif_do_scan(struct kif_proto *p UNUSED)` - Interface scanning. A message requesting interfaces dump is sent to rtnetlink socket. Afterwards the same socket is read until it returns a special message denoting end of data. Each returned message contains info about one network link.

  Afterwards the message is parsed. Information like interface index, name, MTU, flags is extracted. This information is passed to BIRD's core, using function for interface update.

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

  Route attributes like destination network, mask, gateway, origin (for example user added route, route added during boot, BIRD added route) are parsed from received message and passed to 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 is deleted from OS routing table and the new is added to it. If the old is null, only new route is created and likewise if new is null, only old route is deleted. Deletion or creation is done by sending a message with route attributes to rtnetlink socket.

- `nl_async_hook(sock *sk, int size UNUSED)` - asynchronous hook to rtnetlink socket. This hook is called, when kernel sends data to rtnetlink socket, without previous request. This can happen for example when user creates or deletes a route. Kernel then sends message containing info about

created/deleted route. A separate rtnetlink socket is used for receiving these messages.

This socket is added to pool of BIRD sockets during initialization. If kernel sends it message, it is detected by select in main loop. From there, this hook is called and message with route is parsed in the same way as during the scan.

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

1.5.3 BSD

BSD system dependent layer utilizes sysctl system calls and one kernel socket with `PF_ROUTE` domain , `SOCK_RAW` type, `AF_UNSPEC` protocol for OS synchronization.

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

- `void krt_do_scan(struct krt_proto *p UNUSED)` - Routing table scanning. Uses `sysctl` utility, same as interface scanning. Difference is in fifth value of passed management information base, which is set to `NET_RT_DUMP`. Output buffer now contains messages with route info. It is parsed accordingly and retrieved data are propagated to BIRD core.

- `void krt_replace_rte(struct krt_proto *p, net *n, rte *new, rte *old, struct ea_list *eattrs)` - Route replacement. Sends a message with route to previously opened 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` for route creation and to `RTM_DELETE` for deletion.

- `krt_sock_hook(sock *sk, int size UNUSED)` - asynchronous hook of kernel socket. Same as on linux, kernel sends to it info about deleted/created routes. Same socket is used for creating and deleting routes.

1.6 Protocols

This subsection describes protocols that are functional also on HelenOS after BIRD port.

1.6.1 Open shortest path first

Overview of OSPF protocol can be found both on wikipedia [8] and in BIRD documentation [12].

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 [2]) to send and receive OSPF packets. When BIRD is configured to use OSPF protocol on an network interface, it creates one raw socket for each of this interface addresses (if there are multiple addresses assigned to one interface, one socket is created for each). On all of these sockets BIRD sends and receives OSPF packets.

When a hello packet is received, protocol is initialized and eventually the two routers exchange their graph representations of the network. Based on received info each updates its graph network representation. Graph representations are identical on both routers at this point.

From this graph BIRD recalculates routes using dijkstra's algorithm and sends updates to core.

If one router does not hear from the other for a preconfigured period of time, he expects that the neighbor is dead. It sets its protocol state to down (starts the protocol from the beginning) and updates the graph.

1.6.2 Routing Information Protocol

RIP protocol is well introduced both on wikipedia [9] and in BIRD documentation [12].

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, but starts directly advertising its routing table. This may lead to problems covered in evaluation chapter 5.

2. HelenOS

"HelenOS is a portable microkernel-based multiserver operating system designed and implemented from scratch. It decomposes key operating system functionality such as file systems, networking, device drivers and graphical user interface into a collection of fine-grained user space components that interact with each other via message passing. [1]"

This chapter describes the only relevant component of HelenOS for this thesis, which is network stack. At the end of the chapter is also mentioned HelenOS coastline.

2.1 Network stack

HelenOS network stack implementation is split into servers. Servers communicate between themselves and with other applications through IPC. Upper layer servers are acting as clients of lower layer ones. 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 server initialization. Service IDs can be looked up by name or category. When service ID is acquired, client starts IPC communication by connecting to the service, acquiring a session in the process. Clients of network stack servers continue by registering a callback. Callbacks are a way for servers, to send IPC messages to clients asynchronously. Network stack servers use it to pass received network messages. This makes the whole receiving part of network stack asynchronous.

Figure 2.1: HelenOS network stack.

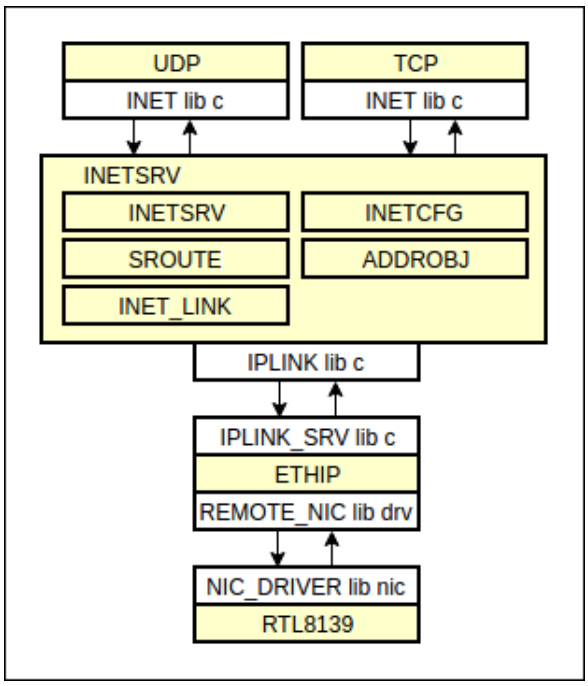


Figure 2.1 shows network stack servers and relations between them. Only

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.

In HelenOS network stack code, network interfaces are referred to as links. Text in this chapter will for this reason also refer to network interfaces as links in some cases. Following sections describes the stack top to bottom.

2.2 Transport layer

There are two servers for two transport layer protocols, UDP and TCP. Next is described UDP server and library, TCP server is not relevant for this thesis.

2.2.1 UDP server and library

UDP server provides one service that accepts requests related to associations and messages.

- Associations - Used to determine client, message is destined to. When association is created endpoint pair is passed from library to server. Endpoint pair structure has following attributes. Local link (IP link service ID), local endpoint (local address and local port) and remote endpoint(remote address and remote port). All parts are optional. If local port is not specified, free port is assigned by server. Association is uniquely identified by endpoint pair (two association cannot have same local port).

- Messages - When sending message, client passes to server data, data length and possibly remote endpoint pair. UDP server creates PDU based on passed parameters and association endpoint pair. Source address and source port are given by association. Destination address and port are given by passed remote endpoint, if its not null. Otherwise they are given by association. One of PDU attributes is also link, set to association endpoint pair link. PDU is converted into datagram and passed to `inetsrv` server.

  When datagram is received from `inetsrv`, it is converted into PDU. Client, that receives the data is determined based on link, local address, local port, remote address, remote port. PDU must match all these parameters of association, in order to be passed to client using that association. Registered callback on client side receives only info about the message. Function assigned to association on client side is invoked with message info as parameter. It is up to implementation of this function, if it retrieves message data from server, based on the passed info.

2.3 Network layer

Implemented by `intesrv` server. This server implements most network stack logic of all servers. The server provides three services. Client connection handler for each service is registered during initialization. The handlers are default, configuration and ping. Next are discussed default and configuration services.

2.3.1 Default service

When client connects to default service, it passes a protocol number. It will then receive all datagrams unpacked from IP packets with this number in IP header. Packets, into which will be packed datagrams passed when sending, will be assigned this protocol number in IP header.

HelenOS network stack is limited by the fact, that `inetsrv` expects only one client per protocol, even though connection of multiple clients passing the same protocol is not prevented. When packet is received from lower layer, datagram unpacked from it is passed to one client only. It is the first client in list of clients, matching protocol number in IP header of the packet.

Currently there are two upper layer servers connecting to `intesrv` server. They are UDP and TCP server, passing UDP and TCP protocol numbers respectively when connecting to `inetsrv` server. This means, when other server or application connects to `inetsrv`, passing TCP or UDP protocol, it will not receive any datagrams, or will cause respective server to stop receiving datagrams.

Client connection handler of this service processes following requests.

- Callback creation.
- Getting source address - in request for source address, destination address is passed. Returned source address, is the address of interface, through which the destination address can be reached. It is looked up according to configured interface addresses and static routes.
- Sending data - when sending, client passes a datagram. Its attributes are link service ID, source address, destination address, type of service, pointer to data and data size. If link service ID is specified, link is given by it, regardless of source or destination address. Otherwise, destination address is used to determine the link. First it is compared to list of interface network addresses. If match is not found, routing table is consulted. If router (gateway) address for given destination is found in routing table, it is still compared to list of interface network addresses. If the link was determined after this process, datagram is packed into packet and passed to `ethip` service.

2.3.2 Configuration service

Service for network configuration, accessed through C library. The service is currently used by `inet` application.

Client connection handler processes requests for changing and getting info about following network configuration.

- Network Interfaces - In `inetsrv` they are represented by `inet_link_t` structure with attributes service ID, service name, session, pointer to IP link, default MTU, MAC address and MAC address validity flag. On network layer, links are discovered by `nconfsrv` server as services of `ethip` and `loopip` servers. After link is discovered, `nconfsrv` passes its service ID to `inetsrv`, using configuration service. At this point `inetsrv` acquires session with `ethip` service, using passed ID, and gets all other attributes of the link through it.

- Network interface addresses - Attributes of each are ID, network address (IPv4 or IPv6 address plus prefix) and pointer to structure representing network interface it is associated with and name.

  When client creates network interface address, it is associated with network interface according to passed name. Service ID of this link is used to pass the address to link layer `ethip` server.

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

- Static routes - represented by structures with destination network address, router (gateway) address and name as attributes. When routing table is consulted for a router address to a destination address, it is simply iterated over all routes, and if match is found, router address is returned.

Server `inetsrv` holds one linked list for links, one for link addresses and one for static routes. All are accessed in a same way. Configuration service provides listing of all IDs from each list. Additionally, info about each of link, link address and static route can be get by ID. Standard way for client to get info about them, is to first retrieve list of IDs. Then are retrieved items of the list by these IDs, one by one. Nothing is locked in between. It can result in trying to get item, that was deleted or not getting item that was added, after getting list of IDs. This must be taken into consideration, when library API is used.

2.4 Link layer

Link layer in HelenoOS consists of NIC drivers and `ethip` server.

Drivers for multiple NIC models are implemented in HelenOS (Figure 2.1 shows RTL8139 driver, but this can be any of the implemented drivers), all of them accessed through the same API. This API is defined in `nic` library. Drivers handle communication between hardware and rest of the network stack. For this thesis is important, that driver must be called in order to enable or disable multicast for a particular NIC. By default, multicast is disabled.

Server `ethip` is responsible for NIC discovery and passing messages between network layer and drivers of discovered NICs. Additionally ARP protocol is implemented here. For this thesis is interesting translation of multicast addresses by this protocol.

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 `hsct.sh` script. The usage is as follows. First is initialized directory where will be POSIX applications built. Initialization includes copying of HelenOS header files and libraries into this directory. In order to install application into HelenOS using `hsct.sh` script,

HARBOUR file for this application must be created. Most commonly, the HARBOUR file specifies URL with sources of ported application, files for patching and how to configure, make and package the application. When build directory is initialized and HARBOUR file is created, application is ready to be installed.

3. Analysis

This chapter first describes overall approach to porting BIRD. BIRD's OS dependencies are analyzed next. Based on these dependencies, current HelenOS state and testing environment capabilities, subset of BIRD functionality to achieve in HelenOS is proposed. Changes both in HelenOS and BIRD required to achieve proposed functionality are analyzed afterwards.

3.1 Testing environment

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

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

3.1.1 Virtual vs real hardware environment

There were two possibilities how to put BIRD into context. First was to construct a real hardware network. The second was a virtual network.

Virtual environment was obvious choice here. Real hardware does not provide any significant advantages.

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

3.1.2 Choice of software simulating the network

Since virtual environment have this many advantages, it was expected that there already existed one, created by BIRD developers. Surprisingly this was not the case. The environment had to be created from scratch.

First were examined possibilities of graphical software simulating network, like GNS3, CORE, Marionnet etc. The chosen tool would be used throughout whole development process. Here are the key requirements on this software.

- capability to connect Qemu VM's running both Unix and HelenOS
- provide means to simulate other network components, at least switches
- fast environment startup, stable during execution

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

This lead to VDE be the tool for virtualizing network. It met all the requirements. The only downside was, that it cannot view any graphical representation of the network. Use of this tool was inspired by thesis HelenOS packet filter [10].

3.1.3 Virtual network participants

It was decided that network will be comprised of VMs running BIRD. Some of the participants could have been running different software, capable of communicating with BIRD. That would however complicate the development process. When all routers in network are running BIRD, it is lot easier to follow debug output and to debug routing protocols on multiple machines simultaneously, since there is no need to get familiar with another routing software.

Next an operating system that will run on VMs was needed. BIRD was ported to BSD and linux systems, so the task was to choose a flavor from one of them.

Core distribution of linux, and its most basic version with only command line, was chosen. It is one of the smallest distributions available. Reasons behind this choice were following.

- fast to boot - this is important, because rebooting will be happening very often during testing.
- 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. This ensures same starting state for testing on each rerun.

3.1.4 Running BIRD

First virtual environment consisted of two cores connected with VDE. After configuring network on both VMs, they were able to communicate.

At this point it was simple to try configuring different protocols with different options, and observe behavior of BIRD on both machines by following debugging output and content of routing tables.

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

Long term strategy was to achieve a state, when one of the cores would be swapped for HelenOS with installed BIRD. Considering routing, behavior of the network should remain the same.

3.2 Porting BIRD

In order to reveal BIRD OS dependencies, coastline build of BIRD was tried. BIRD was configured for cross compilation. The target OS was configured to be linux, so the linux system dependent layer was used during compilation.

3.2.1 Missing dependencies

Build errors revealed all the missing dependencies, since BIRD was now compiled and linked against HelenOS header files and libraries.

Here is list of header files not found by BIRD during compilation. Almost all functions, macros and structures used by BIRD but missing in HelenOS were declared/defined in these header files. Brief description of them is included.

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

`linux/netlink.h` - netlink socket address structure, netlink message and error message structures, macros for manipulating netlink messages. Passed to socket API in order to exchange info about routes and interfaces between BIRD and OS.

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

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

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

`netinet/tcp.h` - 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 important 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, this function operates with. Used to decide which sockets have available data and which are ready to transfer.

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

`sys/uio.h` - input/output vector structure. These structures are part of socket messages.

`sys/un.h` - unix socket address structure. Used to create a kernel socket with specific address, that will be 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 when initializing BIRD 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 also other OS dependencies (like 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 already decide which dependencies will be implemented in HelenOS and which will be replaced by HelenOS alternatives.

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

Second approach was taken. Advantage wes that BIRD could have been compiled and installed into HelenOS quickly. Downside was that some time would be spent on creation of mocks, even entire header files, that would be potentially deleted later.

Almost all mocked macros and structures are related to mocked functions. Macros and structures are passed as parameters, used for preparing input data or parsing output data. All the functions that were missing in HelenOS at the time and were mocked are listed in figures 3.1 and 3.2. All the networking and synchronization with OS 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 Choosing BIRD functionality to implement

Protocols and features in BIRD has their 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. Here is reasoning behind choosing to support or not particular protocols and features. Features, that were functional right after compiling BIRD with mocked dependencies are not mentioned (e.g. configuration file parsing).

Kernel and device protocols

Protocols responsible for synchronization between BIRD and OS. Without functional kernel and device protocols, all other protocols are meaningless. Support is essential.

OSPF protocol

OSPF is the most popular interior gateway protocol. Functionality of this protocol turns HelenOS into OS that can act as a router in most modern autonomous systems. It has the highest priority of all routing protocols, support of this feature is a must.

RIP protocol

Even though usage of RIP protocol in networks nowadays is rare, it does not require any network unrelated dependencies, so it will be included.

BGP protocol

Protocol for exchanging routing information between autonomous systems. Testing this protocol would proof to be difficult (creating and running virtual autonomous systems). It is not ambition of this thesis to turn HelenOS into backbone routing OS. Protocol will not be supported.

BFD protocol

Not a routing protocol itself. This protocol enhances routing protocols by taking care of detection of neighbors upstate. Detection is much faster then the one implemented by routing protocols themselves. Protocol however requires posix threads, which are not implemented in HelenOS mainline, so it will not be supported.

IPv6 support

There are multiple reasons, why supporting IPv6 would be difficult.

HelenOS network stack does not support sending IPv6 messages directly into specified link, which is exactly how BIRD is sending all the messages. "IPv6 implementation still lacks address-scopes support. The routing mecha- nism does not obey address-scopes, which is a non-fatal complication. Link-local addresses are not differentiated by network interfaces. It causes incomplete support of multiple NICs connected to one computer.[14]". Conclusion is that HelenOS is not ready to be used as a router in IPv6 environment with current IPv6 implementation.

Moreover BIRD can only operate over IPv4 or IPv6, depending on what version it is compiled for, so there would have to be ported two instances of BIRD into HelenOS. The same testing set for all the functionality would have to be reapplied in virtual environment configured for IPv6. Implementing IPv6 would require much bigger time frame, so this feature is excluded.

BIRD client support

BIRD client is a minor feature for monitoring and making reconfiguration more user friendly. The client communicates with BIRD over kernel sockets, not implemented in HelenOS. Client will not be included.

3.2.4 POSIX dependencies to implement

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

The requirements can be fulfilled either by implementing mocked dependencies in HelenOS, or by replacing their usage with HelenOS native support in BIRD. It must have been decided which dependencies to replace and which to implement. Dependencies of non supported functionality will remain mocked.

Due to BIRD's design of system dependent parts, three main options presented themselves.

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

The two extremes were rejected and middle path was taken. This meant implementing only network sockets and replacing usage of kernel sockets with HelenOS native alternatives.

If the first approach was to be taken, it would also mean replacing usage of network sockets inside BIRD with HelenOS network stack API. Receiving part of this API is asynchronous, meanwhile BIRD's design requires synchronous communication. Before HelenOS API could be used, it would have to be either converted into synchronous API (in BIRD's system dependent code or in HelenOS) or BIRD would have to be redesigned for asynchronous receiving.

Due to BIRD being single threaded application, redesigning it would be a very difficult task. This leaves the conversion of API, which is much more simple and elegant solution. Since conversion of API is imminent, it can as well be converted to socket API. After HelenOS asynchronous network stack API is converted to synchronous socket API, there is no more need to replace unix system dependent layer. Moreover sockets can be reused for porting other POSIX applications, or be used internally.

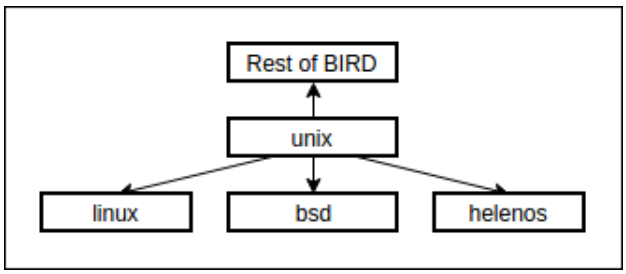
If the other extreme approach was to be taken, additionally to network sockets, kernel rtnetlink sockets would have to be implemented in HelenOS. Here is however no need to convert the API from asynchronous to synchronous.

HelenOS already provides API for getting interface info, getting routing table info and adding and deleting routes from routing table. This API can be consider even more comprehensible and user friendly than the rtnetlink sockets. Conveniently, all the rtnetlink socket functionality is split to BIRD's lowest system dependent layer.

The final decision was to implement network sockets in HelenOS and create new system dependent layer for HelenOS in BIRD, that will be used instead of linux layer, when BIRD is compiled for HelenOS.

Figure 3.3 shows BIRD system dependent layer organization, after this decision. The unix layer calls one of the lower layers (depends on platform, BIRD is compiled for). It also calls rest of the BIRD from main loop. Rest of the BIRD should be system independent (there are a few exceptions).

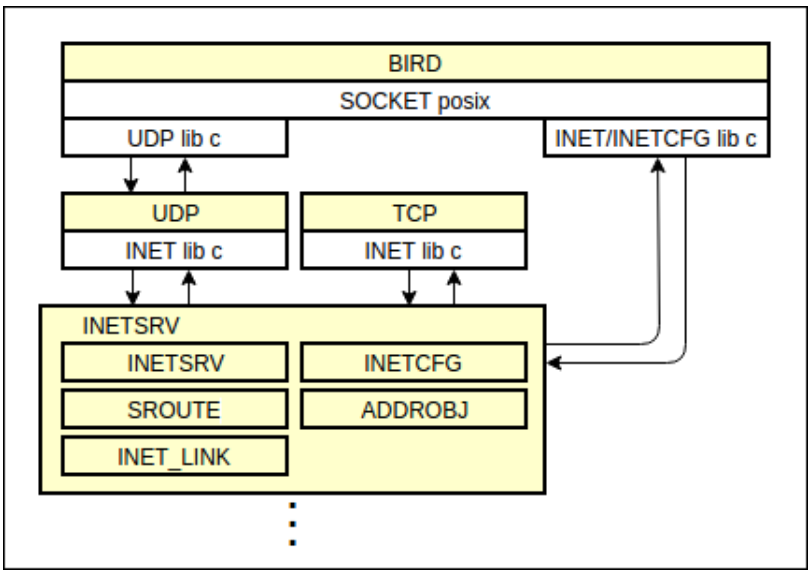
Figure 3.3: BIRD layers regarding OS dependence.



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 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.

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

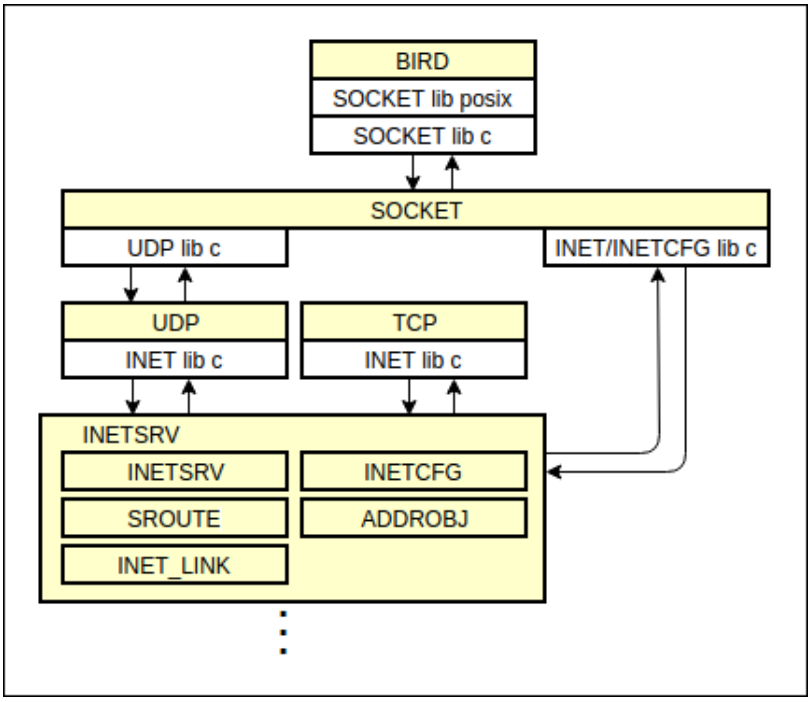


The initial design was to implement sockets purely inside posix library. The posix socket library would use c library network stack API as shown in figure 3.4 and transform the receiving part to synchronous calls.

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 server. This server would be accessed through a c socket library. To BIRD will be presented posix socket library, just like before. Figure 3.5 shows the final design and its incorporation into network stack.

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



3.3.1 Design goals

Sockets on Unix systems are used for multiple purposes and can be configured and adjusted in many ways. Socket implementation, that would be equivalent to unix sockets is neither realistic nor needed. With this in mind these goals were set for HelenOS sockets.

- 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.
- only functionality needed by BIRD will be implemented, but sockets will be easily extensible.

The first goal will be met by creating two sets of socket library functions. One in posix library and one in c library. Posix library will implement no logic, it will

just simply call c library. The c library will handle passing data between user of this library and socket server, no more logic will be implemented here.

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

To achieve the third goal, the design will follow this rule. All parameters passed to socket API will be sent all the way to socket server without any alteration, regardless of if they are used or not. When it comes to passing complex structures, all the needed info will be sent, and exact copies will be recreated on server side.

When all the data arrive on server, it will be decided if implementation of 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 missing implementation.

3.4 Sockets requirements

Previously mocked API shown by figure 3.1, will have to be implemented, except for `listen`, `getsockname` and `sendto`. These three functions are used only by functionality that will not be supported. They will remain as mocks in posix library.

Each network socket used by RIP or OSPF is bound to interface, right after it is created. It is done by setting socket option, where interface name is passed as parameter. Sockets used by RIP are moreover bound to port. It must be possible to bind sockets this way.

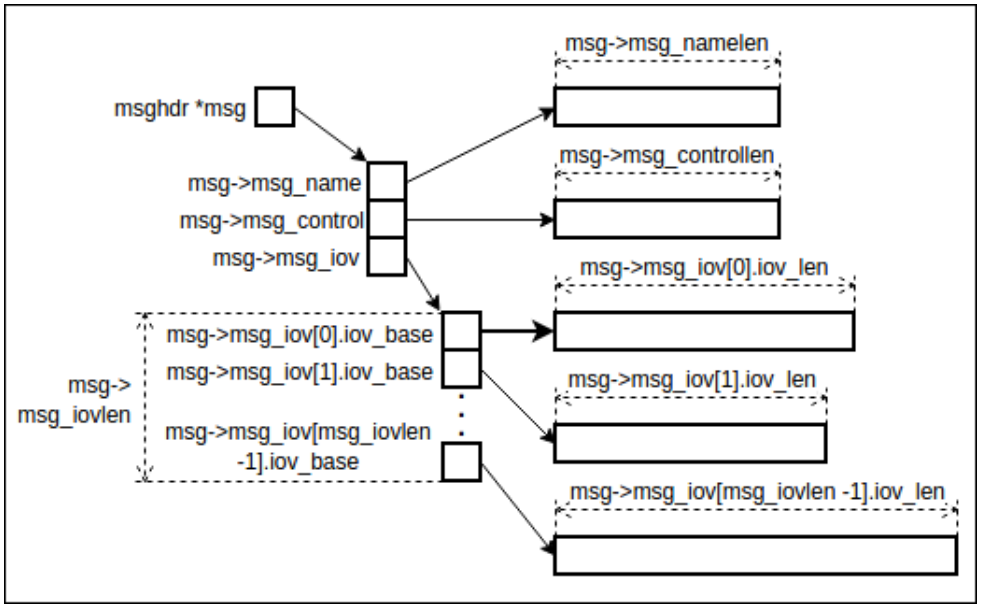
BIRD uses `msghdr` structure both for sending and receiving network messages. Understanding of this structure is crucial to understand how BIRD uses OS to communicate with other network participants.

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

- `void *msg_name` - buffer of size `msg_namelen`, usually contains socket address, describing message destination when sending, or message source when receiving.
- `struct iovec *msg_iov` - pointer to array of input/output vectors. `msg_iovlen` denotes number of vectors in array. Each vector contains buffer `void *iov_base` and its size `iov_len`. The buffers either contain data to send, or will be filled with received data.
- `msg_control` - buffer of size `msg_controllen`. When sending, it can be filled by user with one or more control messages used to additionally adjust sending. When receiving it is filled by socket implementation and contains control messages with additional info about receiving.

Most important is implementation of sending and receiving messages. It will have to deal with translation between `msghdr` structure and form suitable for HelenOS network stack. Translation must respect previous socket set up. This task will be handled by socket server.

Figure 3.6: Message header structure.



Only non blocking implementation of receiving messages is needed by BIRD, but macro `FDISSET` must be able to decide, if there are available data on the socket.

3.5 Additional network stack requirements

After examining, how BIRD uses sockets and how it synchronizes with OS additional requirements on networks stack were found. Requirements on lower layers are described first.

Multicast

Due to both OSPF and RIP using muticast addresses, it must be possible to enable multicast on a NIC associated with a particular link layer name. 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

BIRD periodically synchronizes with OS not only to get new interfaces, interface addresses and routes, but also to get those that were deleted. It must be possible to list these routes and interfaces as well.

Route origin

When routes are scanned, they are treated differently based on their origin. On linux they can be installed for example by administrator, during boot, by system

(when interface addresses are configured), by BIRD etc. Route info must include this attribute.

Routing

There was already existing implementation of routing created in HelenOS packet filter thesis [10]. It was not however included in the mainline repository. After reusing this implementation, it was found out, it can route only one hop. Routing must be functional for any number of hops.

3.5.1 Source address of sent UDP messages

If UDP association is created without specified local address (in local endpoint of endpoint pair), source address of messages sent through this association is null. When sending UDP messages, BIRD only specifies link, through which the messages will be received and sent. If address of this link would be used as local address of association, it would solve the problem, but would create another one. Only messages destined to this address will be received from association. Messages destined to multicast addresses will be filtered out. Following setup must be possible for UDP association. Messages destined to all addresses from given link are received through it, but at the same time local address is added to messages when sending.

Network stack bug fix

When HelenOS was connected to core through VDE, the network stack failed. This happened every time network was configured first in HelenOS and then in core. When configuration was done in reversed order, failures occurred only occasionally. BIRD could not be tested under such circumstances, so the bug must be located and fixed. Initial version of this bug is filed under [3]. Version with detailed description and solution is filed under [7].

3.6 BIRD's HelenOS system dependent layer requirements

The four functions covered in 1.5 must be reimplemented with HelenOS native support. Prerequisites are storing of deleted interface addresses and routes, route origin. For compatibility, this layer must define hooks like reconfiguration, start, shutdown that will be empty or contain just a very simple logic.

4. Implementation

This chapter describes implementation, from point where BIRD is compiled into HelenOS with mocked dependencies and HelenOS is running inside virual environment connected to another VM running core unix.

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. When line debugging information is included, compilation time is multiple times longer. To avoid collisions values of userspace registers EIP, ESP and EBP needs to found out and set in GDB. To add symbol information address of .text section must be manually loaded. This whole process must be repeated on each restart of testing environment.

Fortunately, this is true only for HelenOS. When running BIRD on the core unix, GDB can be installed and used to debug BIRD inside virtualized OS. This way can be at least observed behavior of BIRD running on core. This is especially useful, when data are received from BIRD running on HelenOS.

Additionally tcpdump was used from core unix to monitor traffic coming from HelenOS.

4.2 Socket implementation prequisities

The implementation can be continually tested by running BIRD configured with protocol using socket type currently under development. Only the final implementation is described, phase where logic resided inside posix library is skipped.

There was no need to create a redundant applications both on helenos and core for testing sockets, because BIRD already did what was expected from such application - create and configure sockets, then continually start sending and receiving messages through them. Interference from the rest of the application did not imposed any issues.

The only problem was, that BIRD wasn't sending any data, since synchronization with OS was mocked. That meant scanning interfaces and routes behaved, as if there were none.

It was fixed by more complex mocking of functions scanning interfaces and routes. Now they were returning predefined values. These values were however matching real state of HelenOS configuration used for testing. Writing received routes into routing table was not needed to be functional in order to test sockets.

This way synchronization with OS was simulated and development could focus on sockets.

4.3 Socket posix library

Posix library socket API is there just to satisfy linker when compiling from coastline. There are two exceptions.

One is `close` function, which is the same for VFS and sockets. The highest possible VFS file descriptor is 127. Sockets will use only higher file descriptors, so when posix library `close` is called, it can determine if the file descriptor belongs to a VFS file or a socket and call appropriate c library function for closing. The original `close` function is called for VFS files and newly created `sockclose` function is called for sockets.

Second is `FDISSET` macro. Both raw and UDP sockets are implemented only as non blocking. To determine, if socket has available data, `sockfdisset` function can be called, with socket file descriptor as parameter. `FDISSET` macro just calls this function and `select` function remains mocked. This solution is sufficient for BIRD.

What needs to be mentioned here is that error codes (including socket related) are negative in HelenOS. On the other hand, they are defined as positive numbers in POSIX compliant systems. This is handled by redefining error codes and negating return values of functions in POSIX library. Not even this is needed in case of socket functions because they return -1 on failure and error code is stored in errno variable. POSIX implementation of this variable takes care of returning proper value.

4.4 Socket C library

The only socket logic implemented in libc is passing and receiving data between user of this library and socket server through IPC.

Before client can start communicating with server a session must be created by connecting to it. 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 first socket is created and is used for all subsequent calls.

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

4.4.1 Header files

Some of the structures and macros that previously served only as mocks, now started to be used regularly by c library server. They could have no longer be part of posix library.

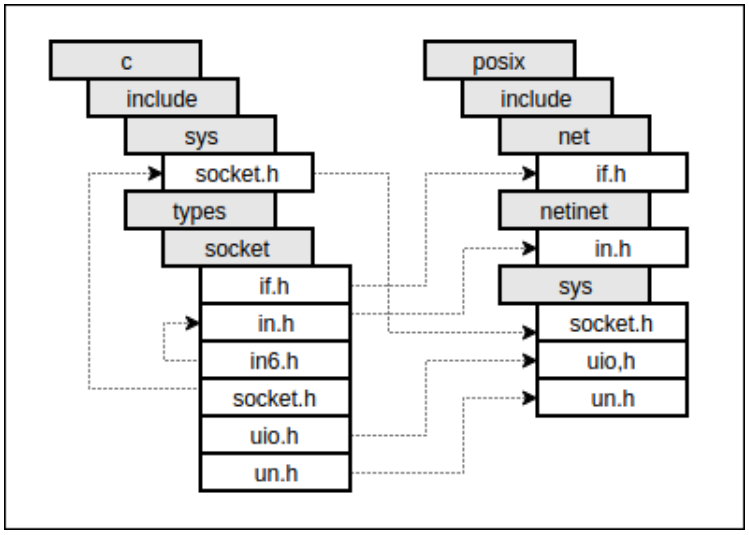
All the header files containing at least one structure or macro used by c library or socket server were moved into c library. The ones with function declarations were split into function declaration only header and macros plus structures header. Under types folder are all header files, that contain only macro and structure definitions. This way, they can be included both from libraries and servers.

Socket related types headers are grouped under socket folder (subfolder of types), even though posix applications expects them scattered in different places

(e.g. netinet/in.h, net/if.h). This problem was solved by adding the same set of headers properly placed in posix library, each just including one header from types.

The figure 4.1 shows source tree of most important posix and c library socket header files and their relations. Arrows mean inclusions.

Figure 4.1: Hierarchical structure of socket related libraries.



Header files containing only mocks were left unchanged in posix library.

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

4.4.2 Socket API

Most of the socket functions are able to pass all the parameters as arguments of initial IPC call, plus one or two calls for passing large data. If any of these function fails at some point, -1 is returned and `errno` is set to appropriate error code. There are two functions that take `msghdr` structure as parameter and therefore the communication is more complex.

The first is `ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags)`. It sends socket file descriptor, number of input/output vectors and flags in initial IPC call. Then all parts of message header are sent as large data in separate calls. First is sent socket address, followed by input/output vectors, each in one call. At the end control message is sent and waiting for return value starts.

The second is `ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)`. Parameters are same as for `sendmsg`, but here the data from `msghdr` structure will not be sent to server, instead it will be filled with data from server. Server only needs to know dimensions of this structure, so it will not send more data than the allocated space. Socket file descriptor socket address length, number of input/output vectors, control message length and flags are sent in initial IPC call. Length of each input/output vector is sent in separate call afterwards. Now server has all the required information, so the function continues with receiving data. All parts of message are received using IPC calls for reading large data. Socket address is received first, next are data into input/output vectors,

each in one call and finally control message is received. After receiving return value from server, total size of received data is retrieved as IPC argument from answer.

4.5 Socket server upper layer

Also called service layer. Same as in other servers, service layer handles client connections. When a connection is established, it waits for client requests in infinite loop, until client hangs up. In comparison to other network stack servers, this server handles all communication in this loop. There are no callbacks to client, because all the communication is synchronous. Each IPC request is handled by one function. Possible requests are request for create, bind, connect, set options, send message, receive message, query if data are ready and close.

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

4.5.1 Sending messages

When server is asked to send message, it retrieves socket file descriptor, number of input/output vectors and flags as arguments of IPC call. Next `msghdr` structure is allocated. All its parts will be allocated and filled as the server continues receiving data from client. Array of vectors is allocated based on size received as IPC argument. Message name (socket address), input/output vectors (each separately), and finally control data are received as large data. For all of them, first is received size of the data, then appropriate part of `msghdr` is allocated and finally data are received into it. When all data are received, `msghdr` structure on server is exact copy of the one passed to library function. Socket is looked up by file descriptor and implementation of sending message is looked up by socket type. If both exist, implementation is called with socket, `msghdr` structure and flags as parameters. Return value of the call is returned to client.

4.5.2 Receiving messages

When receiving message, socket file descriptor, message name length, number of input/output vectors, control data length and flags are retrieved as arguments of IPC call. Structure `msghdr` and all its parts are allocated based on these 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 `sysarg_t` type. Each size is received as large data. When size is received, corresponding input/output vector is allocated. Socket and implementation of receiving message are looked up. Implementation is called with socket, `msghdr` structure and flags as parameters and is expected to fill the structure with data. After the call, contents of `msghdr` structure needs to be sent back to client. They are sent as large data. First is sent message name, followed by input/ output vectors and control data. At the end asynchronous answer with return value and total size of received data as IPC argument is sent to client.

4.6 Socket server lower layer

This layer contains implementation of raw OSPF and datagram UDP sockets. Also unix sockets are mocked here. It handles communication between upper layer and lower network stack servers.

All sockets are stored in one list. Much more appropriate data structure for storing sockets (and many other items in networks stack, like static routes, links, link addresses, etc.) would be a map with O(log n) complexity for adding, deleting and accessing, where file descriptor would be a key and socket structure a value. Unfortunately, HelenOS lacks implementation of map.

Before any of the lower layer functions accesses a socket, whole list is locked. It is not the most efficient solution, but it has no impact on BIRD's performance, unless another application starts using sockets.

Sockets are represented by structures. One structure for each socket type, plus structure with attributes common for all sockets, named `common_socket`, are defined. `common_socket` is at the beginning of each type specific structure, so casting is possible.

One of the common attributes is socket file descriptor, which uniquely identifies socket. In order to generate them, highest socket file descriptor ever assigned and stack of freed socket file descriptors are kept. When there is a need for one, top of the stack is returned. If the stack is empty, highest file descriptor is incremented and return as new socket ID. Highest file descriptor starts at 128. This number is chosen in order to avoid collisions with VFS file descriptors. The highest possible VFS file descriptor is 127.

4.6.1 Raw sockets

Raw sockets on unix are used to send and receive IP packets of higher protocol. The higher protocol is specified by number passed as last argument when socket is created. When IP packet is received, this number is compared with protocol number in IP header. Packet is passed to all sockets matching the number. It is also passed to higher layer of network stack. This means one packet can be received by multiple applications and kernel as well. HelenOS network stack has limitations in this regard, discussed in HelenOS chapter, section 2.3.

Initialization

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

Socket server connects to `inetrv` during initialization, passing 89 (OSPF) as protocol number and callback function for receiving OSPF packets. During this process is created session with `inetsrv`, that will be used for sending messages as well.

When raw socket is created, it is added to list of sockets. BIRD always follows creation of raw socket by binding it to interface, using `setsockopt`. Name of interface, for example `net/eth1` is passed to this function. On Unix, interface indexes are used to uniquely identify interfaces. When a new network interface is discovered by `ethip` server in HelenOS, it creates new service for it and adds

it into IP link service category. Service is assigned unique ID. The service ID can be looked up by interface name. Socket is bound to interface by setting IP link service ID as attribute of socket structure, looked up by interface name. This ID will be used when sending and receiving messages.

Sending messages

Function for sending messages must handle parsing `msghrd` structure into datagram suitable for passing to `inetsrv` server. Datagram's IP link is given by sockets IP link service ID. BIRD always adds only one input/output vector to `msghdr` with all data, so pointer to datagram data is set as base of this vector. Size of the data is set to size of the vector. Destination address is retrieved from `msghdr` socket address, version is set to IPv4. Source address of datagram is more difficult to set properly, because IP link service ID is used to determine which interface will be used to send it. When datagram is sent directly to IP link, none of lower layer servers fills the source address of IP packet generated from the datagram. In this case, source address does not matter to sender, but is crucial for receiver to determine packet origin. Solution is to look it up on socket layer as first IPv4 address configured for IP link, socket is bound to. To find this address, `inetcfg` service of `inetrv` is used. When all the datagram attributes are set, it is sent using `inet_send`.

Receiving messages

Implementation of receiving messages is split into two parts, callback invoked by `inetsrv` and function called when client wants to receive message through raw socket.

Initial implementation of callback function used two conditional variables. When callback was invoked, one would block it, until receiving of message is called. The other would then block receiving function, until message is processed and ready to be received. This was done under false assumption, that each callback invocation runs in a separate fibril and that server invoking the callback does not wait for return value and therefore is not blocked by client.

It was redesigned by adding a list acting as queue of messages to socket structure. When IP datagram is received through callback, all sockets bound to interface, datagram was received from, are looked up. They are looked up in list of sockets by datagram's IP link service ID, which is one of its attributes. Copy of datagram is added to each of these socket's message queue. If no socket is listening on the interface, or there is no more memory to create a copy, datagram is discarded.

When receiving of message is called, it dequeues first message from socket's queue and parses it into `msghdr` structure. All raw sockets are non blocking, so if there is no message in queue, error code is returned. Socket address is parsed first. Its source address is set to datagram source, port is set to zero, and address family is set to `AF_INET`.

Next are parsed actual data. This part is a bit tricky, because sending and receiving messages through raw sockets on Unix is slightly asymmetric. When sending message on Unix, the data do not include IP header. Optionally, IP header can be specified by additional structure, added to message header as

one of control messages. When receiving message, IP header is always part of received data. Server `inetsrv` is on the other hand symmetric and IP header is never part of data. Luckily BIRD only uses first byte of the header to determine packet IP version and length of IP header. These two attributes are reconstructed in first byte. Remaining 19 bytes of header are nulled. Reconstructed header and datagram's data are concatenated into first input/output vector of `msghdr` structure. BIRD always passes `msghdr` with only one vector.

The last to fill is additional info. BIRD expects this part to be filled with one control message. The message type is expected to be packet info and its socket option level to be IP. Data of the message is another structure with actual packet info. This structure contains two attributes, destination address of the packet and index of interface, packet was received through. Index is used to check, that packet was received through expected interface, even though socket was configured to receive only through it. This check seems to be redundant. Destination address is used to distinguish between different addresses configured for one interface, multicast addresses and broadcast addresses. BIRD's protocols process packets differently, based on address they were destined to. Interface index is set to datagram IP link service ID. Destination address is also retrieved from datagram.

4.6.2 UDP sockets

Implementation of UDP sockets is in many ways similar to RAW sockets, but there are some significant differences. During initialization socket server connects to UDP server. Through initial connection is however acquired only reference with session to UDP server and other attributes.

Inititalization

Each UDP socket used by BIRD is bound both to port and to interface. First is set interface with `setsockopt`, same as for RAW sockets. Actual binding happens when `bind` is called by client. Here, reference acquired during initialization is used to create UDP association. Endpoint and callback function are passed. Endpoint specifies IP link and port, association will send and receive messages through. IP link is the one previously set with `setsockopt` and port is retrieved from socket address passed as parameter of `bind`. There is created one association per UDP socket, meaning callback is registered for each. The callback is same function for all, but parameter, it will be invoked with differs. It is a pointer to the socket structure.

Sending messages

When sending messages, the task is to convert `msghdr` structure into four separate parameters in order to send data through association. They are local address, remote endpoint, data and data size. Local address is acquired in the same way as local address for datagram, when sending through raw socket. Remote endpoint address and port are retrieved from `msghdr` socket address. Data are set to base of first input/output vector and their size to size of this vector.

Receiving messages

Callback function and function for receiving messages cooperate in a same way as in raw socket implementation. Callback is adding UDP messages to socket queues, and receive function is consuming them. There is however no need to look up socket, message is destined to, because destination socket is passed as callback parameter.

When receiving, message is dequeued. It needs to be converted into `msghdr` structure. Socket address is filled with remote endpoint address and port. Remote endpoint is part of dequeued message. First input/output vector is filled with message data and its size is set to message size.

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

Unix sockets

Kernel socket on unix layer in BIRD is used to detect, if there is already an instance of BIRD running. This is done by connecting to a specific socket unix path. If another socket is bound to this path and connection is successful, BIRD concludes there is another instance running and shuts down. If not, then it binds its socket to this address. This functionality is mocked on server, because it must be possible to distinguish between binding to network socket and unix socket. This can be done only after looking up socket structure by socket file descriptor.

4.7 Additional network stack changes

This section describes implementation of requirements from 3.5. These requirements were tried to be implemented in least invasive way possible.

4.7.1 Multicast

In order to make it possible for BIRD to allow multicast on driver level, driver service ID was added to network layer link info. Link info is accessed through `inetcfg` service of `inetsrv` server. This server acquires driver service ID from `ethip` server in separate IPC call during link opening, same as other info like MTU, MAC address, etc.

Translation of two multicast addresses were added to `ethip` ARP protocol. They are OSPF and RIP multicast addresses, specified by RFC [5] and [6] respectively. Both are translated to ethernet addresses locally, 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 local addresses.

4.7.2 Deleted interface addresses and routes, route origin

Two additional lists were created for storing deleted interface addresses and routes. When interface address or route is deleted, it is moved from list of regular items into list of deleted items. When new interface address or route is created it is added in the same way as before, but additionally it is looked up in list

of deleted items. If match is found, item is removed from this list and deallocated. Matching criteria includes all attributes except name for both addresses and routes.

API to listing routes and interface addresses needed a change as well. This library API consist of four functions. Getting list of all interface addresses ID's, getting all routes ID's, getting interface address info based on ID and getting route info based on ID. To each of this functions was added parameter specifying, what status should the listed addresses or routes have. The status can be active or deleted. When IPC request arrives on server, implementation will return addresses or routes from list of regular or deleted items, based on status.

Route origin is added as another attribute of route info. API for creating routes was extended by a parameter to set this attribute. It is up to user of library to fill it properly.

4.7.3 Routing

Implementation of routing from thesis Packet filtering took advantage of the fact, that HelenOS packets with local origin were already routed. When a foreign packet is received, and `inetsrv` decides it is not destined to one of local addresses, it is routed the same way as packets with local origin, instead of being discarded.

Problem with this implementation was, that packet was forwarded only if its destination address matched network address of one of the interfaces. This was the cause of HelenOS routing only one hop. This check was unnecessary, because if there is no match, routing table is consulted. After removing the check, the issue was fixed.

4.7.4 Source address of sent UDP messages

In UDP library parameter specifying source address was added to function for sending messages through UDP association. It is passed, like other parameters to server. Server handles this parameter similarly to remote endpoint parameter. If the passed local address is null, local address specified by local endpoint is uses (it can still be null, in which case also source address of the message will be null). Otherwise the passed address is used as source address of the message. Now only link can be specified, when association is created, and source address can be supplied with each message. It also covers case, when link address changes.

4.7.5 Network stack bug fix

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

There was found a typo error in assert in UDP server. When the server received UDP packet, it tried to look up association, packet was destined to. Association is most commonly looked up by port. If the association was not found, server crashed on assert, instead of just discarding the packet.

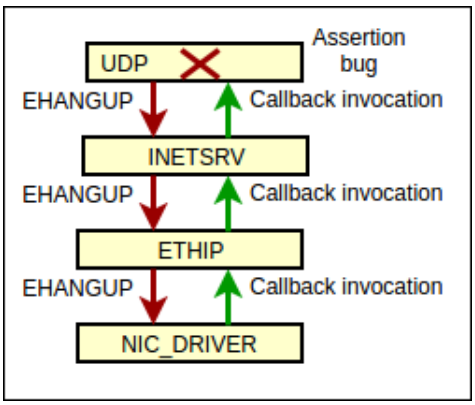
The bug was discovered, because the other network participant was broadcasting some UDP packets and there was no consumer for them in HelenOS.

The bug was easily removed by fixing the typo in assert, but the implication is that failure of one high layer server caused harm to all network stack servers. This should not happen on a microkernel system architecture. The whole point of splitting bigger components, like network stack, into smaller modules is to prevent failures like this. It is also in conflict with statement in text introducing HelenOS.

"A failure or crash of one component does not directly harm others. HelenOS is therefore flexible, modular, extensible, fault tolerant and easy to understand. [1]"

Possible cause is the way errors are propagated in network stack. When this bug occurred, and UDP server crashed, `EHANGUP` error code was propagated as return value of each callback invocation, down to the driver as illustrated by figure 4.2.

Figure 4.2: Hangup propagation.



After receiving `EHANGUP`, driver failed to invoke callback of `ethip` for all subsequently received messages, which lead to network stack functional only for sending.

4.8 BIRD's HelenOS system dependent layer

Synchronization with HelenOS is implemented using `inetcfg` library. This layer is implemented in HelenOS coastline and is added to BIRD during coastline build process.

4.8.1 Interface scanning

Interface scanning is implemented by getting info about all links and link addresses. From each link info are extracted interface name, link service ID and MTU. Link service ID is used as interface index. Temporary BIRD interface structure is created with these attributes and passed to core interface update. Interface addresses are acquired in two steps. First are listed active, then deleted addresses. From each interface address info are extracted interface address, interface address prefix length and link service ID as interface index. Other attributes

(e.g. scope, broadcast address, network address) are calculated next. For active addresses, core address update is called. If deleted address is listed, core address deletion is called.

4.8.2 Routing table scanning

Routes are scanned by listing route info of all routes. First are listed active, then deleted routes. Destination address, destination address prefix length, router (gateway) address and route origin are acquired from each route info. On linux, deleted routes are handled by reading rtnetlink kernel socket and invoking hook as described in 1.5. This is substituted in HelenOS by storing deleted routes and processing them during route scan. Deleted routes are, unlike deleted interface addresses, not stored in linux.

4.8.3 Creating and deleting routes

BIRD internal structure for routing table entry is passed to function for creating and deleting routes. Attributes destination address, destination address prefix length and router (gateway) address of this structure are used. Route in HelenOS routing table matching these attributes is looked up. If match is not found and route is being created, new route with these attributes is added to routing table. Matching route is looked up in order to avoid duplicates. If route is being deleted and match is found, the route is deleted from HelenOS routing table, if there is no match, function reports error.

5. Evaluation

5.1 Environment setup

Implementation was tested in virtual environment proposed in 3.1. Two variants of this environment were used. Appendix B contains description how to run both. When environment is running, network must be configured and BIRD's started. This must be done on each node separately. Four scripts are prepared in each image for this task. They are named `basic-ospf-bird.sh`, `basic-rip-bird.sh`, `triangle-ospf-bird.sh` and `triangle-rip-bird.sh`. Script syntax is different on HelenOS and on cores, but the logic is same. Usage of scripts differs based on which set of tests is performed. They can be found in home directory on all images.

Figures (5.1 and 5.3 in test description are illustrating network configurations, after these scripts are executed. During OSPF tests, network configuration is modified multiple times to simulate network changes. Modification is done by deleting and restoring network addresses on interfaces. These changes are illustrated by figures 5.2 and 5.4. States are described by simplified images of topology from figure 5.1 in figure 5.2 and from figure 5.3 in figure 5.4. Red lines in figures 5.2 and 5.4 means, that there is no address configured for router interface, where the line starts.

Alternatively, the network changes can be simulated by bringing interfaces down/up on Cores. HelenOS does not support this functionality.

When a change is made to network, some period of time needs to elapse, before BIRD's exchange all the info and update routing tables. Waiting twenty seconds 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 twenty seconds, results are verified by examining routing tables on all routers. Additionally, from each node are pinged interfaces of other nodes, that are part of destination networks of newly added/deleted routes.

5.1.1 BIRD configuration

When BIRD is started by one of the scripts, configuration file is passed as argument. Scripts for testing OSPF protocol are passing `ospf-bird.conf` and `rip-bird.conf` is passed by scripts for RIP protocol testing. These configuration files are also located in home directories on all images. BIRD configurations are same for basic and triangle topology. This is possible because RIP and OSPF are configured for all interfaces. Side effect is that the protocols will send messages also to interfaces, where no router is connected on other side, but this poses no issues.

In `ospf-bird.conf` BIRD is configured to scan routing table every twenty seconds and export all routes to core, by setting kernel protocol scan time to twenty and export to all. Interface scanning is configured to occur every five seconds by setting device protocol scan time to five.

OSPF protocol is configured for one area including all interfaces. Each interface is configured with the same options. Next are described these options.

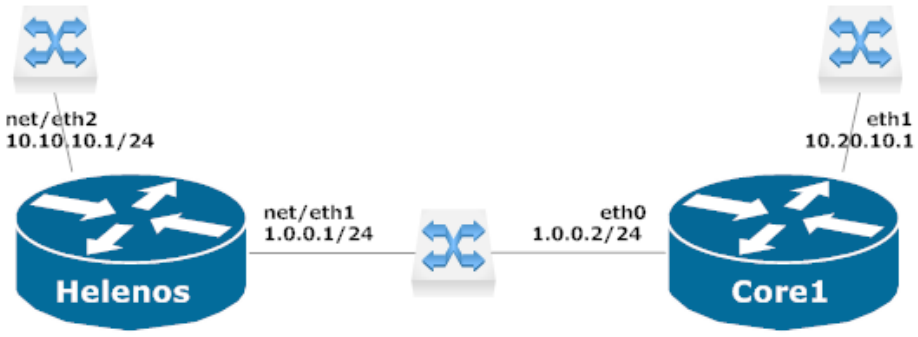
Type is set to point to point, since each interface is connected to none or exactly one other router. Period for sending hello packets is set to three seconds, retransmission after not getting acknowledgement to two seconds, waiting after start for adjacency build to five seconds, number of seconds before proclaiming neighbor dead after not hearing from him to eight and mode to multicast. OSPF configuration is same on Cores and HelenOS.

In `rip-bird.conf` kernel and device protocols are configured same as in `ospf-bird.conf`. The only exception is learn option for kernel protocol. RIP is only able to send content of routing table, so the routes must be added manually. This is not needed for OSPF, because OSPF constructs graph based on scanned interfaces and info it receives from other routers, then calculates routes from this graph. After routing table scan, routes added by admin are propagated to core only if learn option is enabled. On unix, routes are added automatically when interface addresses are configured, but BIRD ignores these routes in all cases, even when learn option is enabled. HelenOS adds no routes when interface address is added. Manually added routes are equally needed in Helenos and Cores. Alternative to manually added routes and enabling learn option, could be routes added in static protocol, resulting in same behavior. Next is configured RIP protocol for all interfaces. Period for sending routing table content is set to five seconds and mode to multicast.

5.2 Basic topology tests

This section describes tests performed in the first variant of virtual environment. It simulates most basic topology consisting of just two routers and three switches, illustrated by figure 5.1. This topology was used during development. It is the simplest setup required to verify that tested protocol is functional, at least on one interface. Following tests will assume, basic environment is started, as described in appendix B.

Figure 5.1: Basic topology.



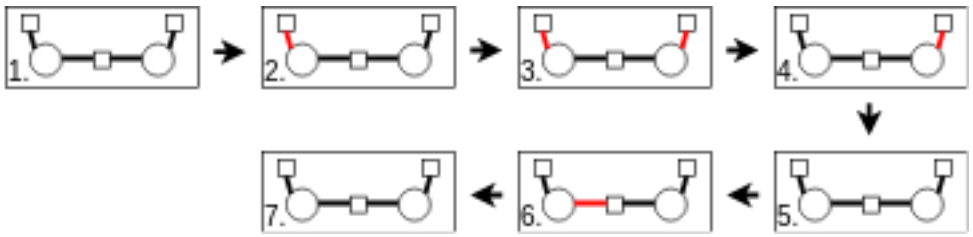
5.2.1 OSPF test

Test is initialized by running script `basic-ospf-bird.sh` both on Helenos and on Core1. Order of execution is arbitrary. Each script configures two interface

addresses and starts BIRD with OSPF configuration. Result is network address assignment to interfaces as displayed by figure 5.1.

Figure 5.2 shows network states during the test. States are numbered one to seven.

Figure 5.2: Basic topology OSPF test.



When network is in initial state (state 1.), each router knows about the network in between HelenOS and Core1 and about the other network it is connected to. Neither however knows about network behind the opposite router. That is Helenos does not know about 10.20.10.0/24 and Core1 does not know about 10.10.10.0/24. When BIRD is started on both, routes to these networks appear in respective routing tables.

Next step of the test is to evaluate behavior, when network changes. First is deleted address on `net/eth2` interface in HelenOS (state 2.). BIRD running on HelenOS notices, that the interface is not configured and therefore no longer part of the network. This change is propagated to BIRD running on Core1. Result is deletion of route into network behind this interface from Core1 routing table. Likewise, when address is deleted next from `eth1` interface in Core1 (state 3.), route to network behind this interface is deleted from Helenos routing table.

Deleted interface addresses are restored on Core1 (state 4.) and on Helenos(state 5.) next. The routing tables goes back to state before the addresses were deleted. Each contains one route.

Next is deleted `net/eth1` interface address in Helenos (state 6.). This cuts the communication between the two routers. When address is deleted, Helenos knows about the change instantly and removes route to network behind Core1. On the other hand, Core1 must wait eight seconds, after last received packet from Helenos, before it can conclude it is dead. The route to network behind Helenos is deleted shortly after this interval elapses.

Last step is to restore deleted address (state 7.), reestablishing communication between the two routers. One route appears in each routing table, same as before the communication was cut off.

5.2.2 RIP test

The test starts with running `basic-rip-bird.sh` in Helenos and Core1. This script configures network, same as the script for OSPF test, adds one route to network behind the router into routing table and starts BIRD with RIP configuration.

The script adds route with destination network 10.10.10.0/24 and gateway 10.10.10.1 in Helenos and route with destination network 10.20.10.1 and gateway 10.20.10.1 in Core1.

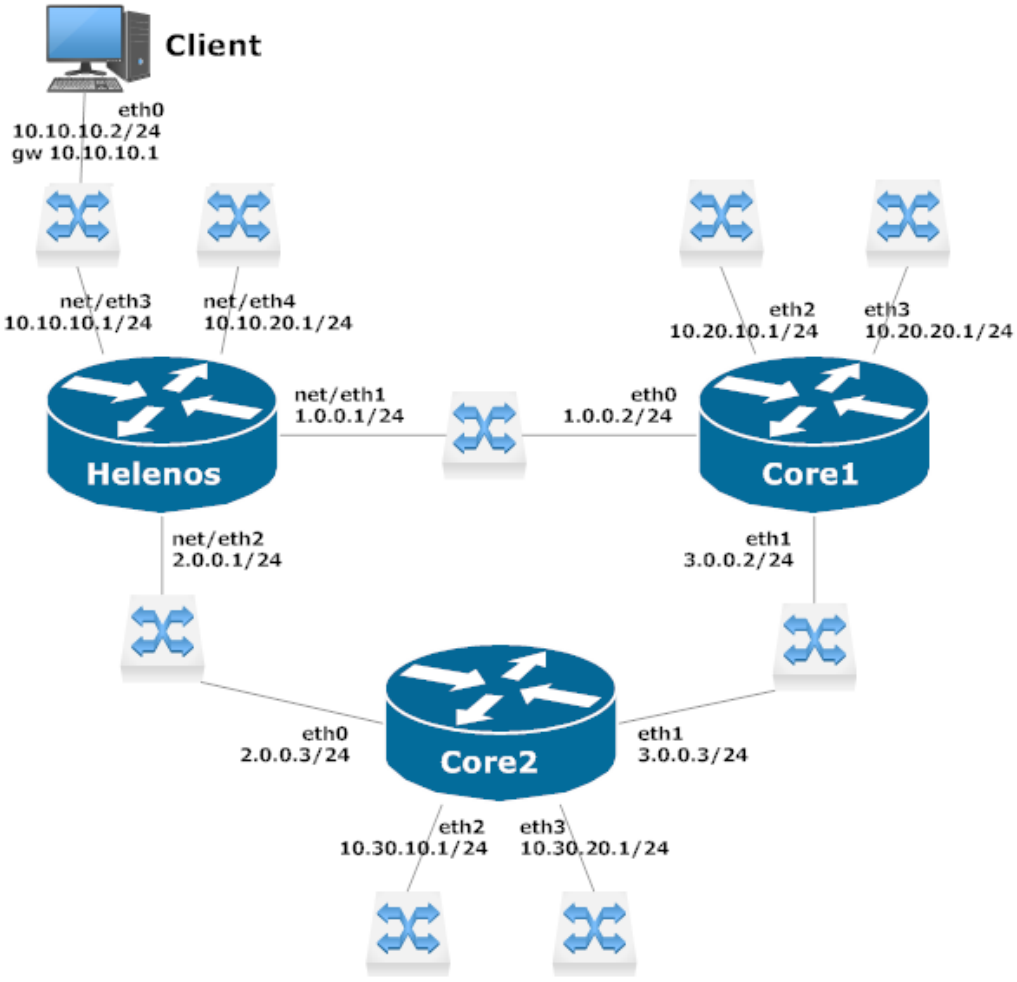
When BIRD is started, it is expected, that these routes are exchanged between the two routers and each of them has two routes in routing table.

It is the only step of the test, because when a route is deleted on one router, and updated content of routing table is sent to the other, BIRD that receives update does not delete it from OS routing table. This is however not an HelenOS related issue. When the network consisted of VM's running Cores only, the behavior was the same.

5.3 Triangle topology tests

Another set of tests is performed in virtual environment simulating more complex topology, shown in figure 5.3.

Figure 5.3: Triangle topology.



It consist of three routers, nine switches and one VM simulating endpoint user, named Client. One router is running HelenOS and two are running Core linux. Switches connecting routers will be called inner switches and all other switches will be called outer switches. Every pair of routers is connected, forming a triangle. Each router has four interfaces. Two are connected to inner switches, and two to outer switches, simulating two networks behind each router. Client is

also running Core linux. It can be used to verify, that Helenos is routing correctly by pinging any interfaces in the network, at any stage of the tests.

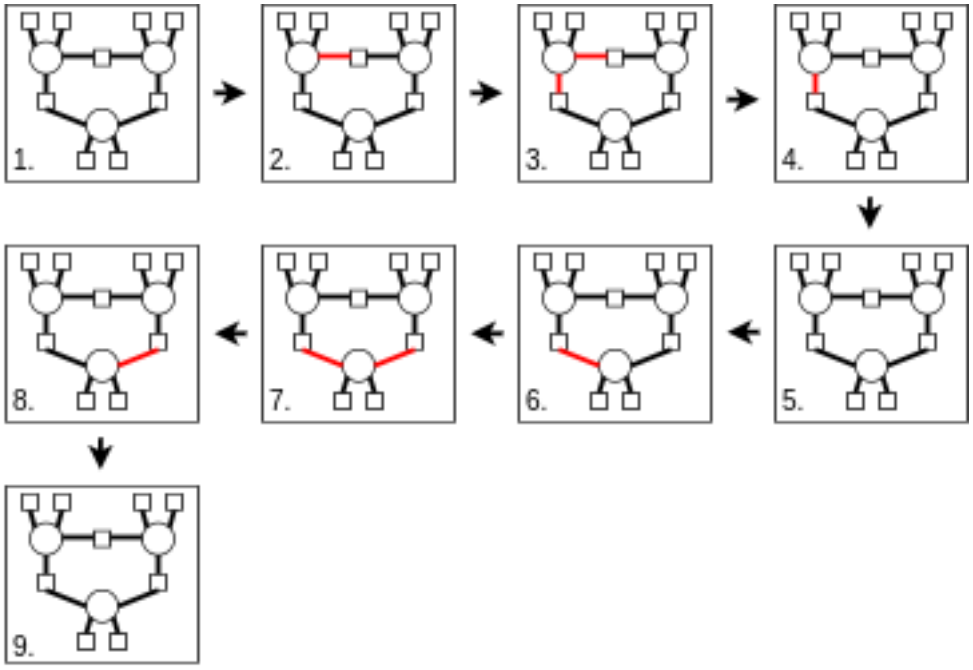
Following tests will assume, triangle topology environment is started, as described in appendix B.

5.3.1 OSPF test

Test is initialized by running `triangle-ospf-bird.sh` in Helenos, Core1 and Core2. Execution of these scripts results in network interface address configuration on all three routers as shown by figure 5.3 and start of BIRD with OSPF configuration.

To test OSPF protocol, network state is changed eight times. States are numbered one to nine, shown by figure 5.4.

Figure 5.4: Triangle topology OSPF test.



When network is configured (state 1.), but there was no exchange between BIRD's yet, each router is aware of four networks around (one behind each interface). There are however five networks it is unaware of, two networks behind each of other two routers and network connecting them. The network is symmetric from each router point of view, so this is true for all of them. After OSPF exchange between BIRDs on all three routers is concluded and all BIRDs are synchronized with OS, routing table of each router contains five routes to the five networks. Tables 5.1, 5.2 and 5.3 shows routing table contents on Helenos, Core1 and Core2 respectively at this point. Routes added by OS, when network interface addresses are configured are not included. In each routing table, gateway of route to network between other two routers can be any of the two, because distance to this network is same through both. Result depends on from which of the two is info about the network transfered through OSPF first.

Table 5.1: Helenos routing table after initial OSPF exhange.

Route index	Destination network	Gateway(Router)	Interface
1.	3.0.0.0/24	1.0.0.2 or 2.0.0.3	-
2.	10.20.10.0/24	1.0.0.2	-
3.	10.20.20.0/24	1.0.0.2	-
4.	10.30.10.0/24	2.0.0.3	-
5.	10.30.20.0/24	2.0.0.3	-

Table 5.2: Core1 routing table after initial OSPF exhange.

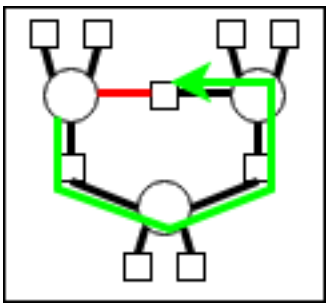
Route index	Destination network	Gateway(Router)	Interface
1.	2.0.0.0/24	1.0.0.1 or 3.0.0.3	eth0 or eth1
2.	10.10.10.0/24	1.0.0.1	eth0
3.	10.10.20.0/24	1.0.0.1	eth0
4.	10.30.10.0/24	3.0.0.3	eth1
5.	10.30.20.0/24	3.0.0.3	eth1

Table 5.3: Core2 routing table after initial OSPF exhange.

Route index	Destination network	Gateway(Router)	Interface
1.	1.0.0.0/24	2.0.0.1 or 3.0.0.2	eth0 or eth1
2.	10.10.10.0/24	2.0.0.1	eth0
3.	10.10.20.0/24	2.0.0.1	eth0
4.	10.20.10.0/24	3.0.0.2	eth1
5.	10.20.20.0/24	3.0.0.2	eth1

Next, address is deleted from `net/eth1` interface in Helenos, cutting the communication between Helenos and Core1 (state 2.). BIRDs make following changes in routing tables. In Helenos, routes with gateway 1.0.0.2 (indexed 2., 3. and possibly 1.) are replaced by routes to the same destinations, but with gateway 2.0.0.3. Additionally one more route is added, destined to 1.0.0.0/24. Path to this network was previously given by interface network address. Now this network is only behind Core1 router. The route is illustrated by green arrow in figure 5.5.

Figure 5.5: New route.



In Core1, routes with gateway 1.0.0.1 (indexed 2., 3., and possibly 1.) are replaced by routes with same destinations, but with gateway 3.0.0.3. Here BIRD adds no additional route, because the network 1.0.0.0/24 is still accessible directly through `eth0` interface.

In Core2 routing table, BIRD replaces route destined to 1.0.0.0/24 by route with same destination, but with gateway 3.0.0.2 in case the gateway was previously 2.0.0.1. Otherwise the routing table remains unchanged.

Deletion of address from `net/eth2` in Helenos by user follows, leaving it cut off from other two routers (state 3.). Result is deletion of all routes in Helenos routing table and deletion of routes into networks behind Helenos from core1 and core2 routing tables (indexed 2. and 3. in both) by BIRDs.

Address is restored on `net/eth1` interface in Helenos by user (state 4.), resulting in following changes, compared to previous state. BIRD adds six routes to Helenos empty routing table. They are one route to network between Core1 and Core2 (3.0.0.0/24), two routes to networks behind Core1 (10.20.10.0/24 and 10.20.20.0/24) and three routes to networks behind Core2 (10.30.10.0/24, 10.30.20.0/24 and 2.0.0.0/24). All have the same gateway, 1.0.0.2. To Core1 and Core2 are added by BIRD routes to networks behind Helenos (10.10.10.0/24 and 10.10.20.0/24). In Core1 their gateway is 1.0.0.1 (Helenos interface) and in Core2 it is 3.0.0.2 (Core1 interface).

Next address is restored on `net/eth2` by user as well (state 5.). This puts the network into state identical with initial state, so BIRDs also return routing tables into initial state, shown in tables 5.1, 5.2 and 5.3.

User continues by deleting `eth0` address in Core2 (state 6.). In Helenos BIRD replaces routes with gateway 2.0.0.3 (indexed 4. and 5.) by routes with same destination, but gateway 1.0.0.1. If in Core1 route to 2.0.0.0/24 has gateway 3.0.0.2, BIRD replaces it by route with same destination but gateway 1.0.0.1. In Core2 BIRD replaces routes with gateway 2.0.0.1 by routes with same destination but gateway 3.0.0.2 and route to 2.0.0.0/24 with gateway 3.0.0.2 is added.

When `eth1` address in Core2 is deleted by user as well (state 7.), leaving Core2 cut off from other two routers, BIRD deletes all routes, it previously added, from Core2 routing table. BIRDs on Helenos and Core1 deletes routes into networks behind Core2 (indexed 4., 5. in both tables).

Address is restored on `eth1` interface in Core2 (state 8.) by user. At this point, the only connection between Core1 and Core2 is through Helenos. BIRD on Helenos now have to delegate info from Core1 to Core2 and the other way around. Six routes are added to Core2 routing table by BIRD. They are one route to network between Helenos and Core1 (1.0.0.0/24), two networks behind Helenos (10.10.10.0/24, 10.10.20.0/24) and three networks behind Core1 (10.20.10.0/24, 10.20.20.0/24 and 3.0.0.0/24) all with gateway 2.0.0.1. Both in Helenos and Core1 BIRDs add two routes to networks behind Core2 (10.30.10.0/24 and 10.30.20.0/24). In Helenos the gateway is 2.0.0.3 and in Core1 it is 1.0.0.1.

Test is concluded by returning address to `eth2`, putting network into initial state, and again BIRDs restore routing tables into state shown by tables 5.1, 5.2 and 5.3.

5.3.2 RIP test

The test starts by running `triangle-rip-bird.sh` scripts in Helenos, Core1 and Core2. These scripts configure network interface addresses, as shown by figure 5.3, same as the script for OSPF triangle topology test. Then each adds two routes into networks behind outer switches on each router and starts BIRD with

RIP configuration. When BIRD's are started, each router receives four routes from the other two routers. Contents of routing tables of Helenos, Core1 and Core2 at that point are shown in tables 5.4, 5.5 and 5.6 respectively. First two routes in each are added by initial script. Other four are added by BIRD.

Table 5.4: Helenos routing table after initial RIP exhange.

Route index	Destination network	Gateway(Router)	Interface
1.	10.10.10.0/24	10.10.10.1	-
2.	10.10.20.0/24	10.10.20.1	-
3.	10.20.10.0/24	1.0.0.2	-
4.	10.20.20.0/24	1.0.0.2	-
5.	10.30.10.0/24	2.0.0.3	-
6.	10.30.20.0/24	2.0.0.3	-

Table 5.5: Core1 routing table after initial RIP exhange.

Route index	Destination network	Gateway(Router)	Interface
1.	10.20.10.0/24	0.0.0.0	eth2
2.	10.20.20.0/24	0.0.0.0	eth3
3.	10.10.10.0/24	1.0.0.1	eth0
4.	10.10.20.0/24	1.0.0.1	eth0
5.	10.30.10.0/24	3.0.0.3	eth1
6.	10.30.20.0/24	3.0.0.3	eth1

Table 5.6: Core2 routing table after initial RIP exhange.

Route index	Destination network	Gateway(Router)	Interface
1.	10.30.10.0/24	0.0.0.0	eth2
2.	10.30.20.0/24	0.0.0.0	eth3
3.	10.10.10.0/24	2.0.0.1	eth0
4.	10.10.20.0/24	2.0.0.1	eth0
5.	10.20.10.0/24	3.0.0.2	eth1
6.	10.20.20.0/24	3.0.0.2	eth1

When this test was performed multiple times, in some cases the four routes added by BIRD had the same gateway on one of the routers. It happened always on a router, where BIRD was started last. It is a bug, because the shortest distance to two networks behind each other router, is through that router, and therefore it should be the gateway for the pair. This bug occurred even when Helenos was swapped for another Core unix, meaning this is a bug in BIRD's RIP protocol and is not related to HelenOS.

5.4 Test results

OSPF protocol behaved in both environments as expected, correctly reacting to all network changes. The changes included network paths alterations, solicitation and reconnection first of HelenOS and then of Core. In the process, first Core and then HelenOS were made a router between other two routers.

Two issues were found with RIP protocol. When routes are deleted on one router, other routers does not react to this change. In more complex environment, one of the routers did not choose the shortest possible routes to two of the networks. Both of these problems are however not HelenOS related. RIP protocol was redesigned in more recent version of BIRD so the issues were not reported.

These tests prove, that raw sockets, UDP sockets and synchronization with OS are fully functional and BIRD is running in HelenOS, as expected.

Conclusion

Goals of the thesis were partially achieved. After the port, HelenOS provides BIRD with support of two main routing protocols, OSPF and RIP, over IPv4. HelenOS with running BIRD is ready to be used as a routing OS in IPv4 environment.

This goal was achieved by implementing network sockets, required by the two protocols, in HelenOS, improving HelenOS network stack and extending BIRD with new system dependent layer specific for HelenOS. Sockets were implemented in new socket server. Socket API that uses this server was added to c and posix libraries. This design respects HelenOS microkernel multiserver architecture. Even though created socket implementation only includes functionality needed by BIRD, it is easily extensible and ready to be used by other POSIX and HelenOS native applications.

Virtual network environment for testing was created with combination of Qemu and VDE. This environment is also reusable for testing other HelenOS networking related features. Two topologies are available for testing. Network configuration in this environment was dynamically changed multiple times during tests, in order to change network paths. In case of OSPF protocol, all changes were correctly communicated between HelenOS and other routers, resulting in proper configuration of all routing tables in all states and proving implementation is functional. In case of RIP protocol tests, only addition of new routes was reflected by all routing tables. In some cases, routes in one of the routing tables were not the shortest possible. However, these issues are not related to HelenOS, and the tests prove that HelenOS support of this protocol is functional.

Future work

There are still BIRD functionalities not supported in HelenOS.

To support IPv6, HelenOS network stack needs multiple IPv6 related upgrades. Most of them are discussed in thesis IPv6 for HelenOS [14].

Part of another thesis, Port of QEMU to HelenOS [13], dealt with implementation of POSIX threads. Unfortunately, the implementation is not merged into HelenOS trunk. If merge of this branch into trunk happens in future, it should be simple to support BFD protocol.

If there would be a requirement on HelenOS to act as a router between autonomous systems, sockets are ready to be extended to support TCP protocol and therefore BGP.

New versions of BIRD with new features are released continually. 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.

Bibliography

[1] Helenos. URL http://www.helenos.org/.

[2] Protocol numbers. URL https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[3] Ping not working. URL http://www.helenos.org/ticket/667#.

[4] Rfc1112, . URL https://www.ietf.org/rfc/rfc1112.txt.

[5] Rfc1247, . URL https://tools.ietf.org/html/rfc1247.

[6] Rfc2453, . URL https://tools.ietf.org/html/rfc2453.

[7] Udp crashing network stack. URL http://www.helenos.org/ticket/672.

[8] Open shortest path first, . URL https://en.wikipedia.org/wiki/Open_Shortest_Path_First.

[9] Routing information protocol, . URL https://en.wikipedia.org/wiki/Routing_Information_Protocol.

[10] Jan Buchar. Helenos packet filter. URL http://www.helenos.org/doc/theses/jb-thesis.pdf.

[11] Ondřej Filip, Pavel Machek, Martin Mareš, and Ondřej Zajíček. Bird programmer's documentation, . URL ftp://bird.network.cz/pub/bird/bird-doc-1.5.0.tar.gz.

[12] Ondřej Filip, Pavel Machek, Martin Mareš, and Ondřej Zajíček. Bird user's guide, . URL ftp://bird.network.cz/pub/bird/bird-doc-1.5.0.tar.gz.

[13] Jan Mareš. Port of qemu to helenos. URL http://www.helenos.org/doc/theses/jm-thesis.pdf.

[14] Antonín Steinhauser. Ipv6 for helenos. URL http://www.helenos.org/doc/theses/jb-thesis.pdf.

List of Abbreviations

API - Application Programming Interface
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 `lp:~galfy/helenos/bird-port-mainline` containing HelenOS mainline with implementation described it this thesis, merged with mainline trunk revision number 2592. This is revision number 2587 of this repository.
- HelenOS/mainline/image.iso - Pre-built image of HelenOS.
- HelenOS/coastline - Clone of repository `lp:~galfy/helenos/bird-port-coastline` containing HelenOS coastline with implementation described it this thesis, branched from coastline trunk revision number 119. This is revision number 131 of this repository.
- HelenOS/topology - Clone of repository `https://StanislavGalfy@bitbucket.org/StanislavGalfy/topology.git` containing images and script, used to set up virtual environment for testing. This is master branch after commit. `f05da3` of this repository.
- thesis-latex - latex sources of this text.

B. Compiling and running

Prerequisites:

- VDE2
- Qemu - must be compiled with `--enable-vde` option. When using package manager, in some cases qemu without enabled vde is installed. To ensure this option is enabled, install qemu with script included in HelenOS mainline `contrib` folder.
- GIT - optional, for getting sources from online repositories
- Bazaar - optional, for getting sources from online repositories

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

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

```
mkdir ∼/HelenOS
bzr branch lp:∼galfy/helenos/bird-port-mainline
    ∼/HelenOS/mainline
bzr branch lp:∼galfy/helenos/bird-port-coastline
    ∼/HelenOS/coastline
git clone https://StanislavGalfy@bitbucket.org/
    StanislavGalfy/topology.git ∼/HelenOS/topology
```

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

Next step is to initialize directory for coastline build, as instructed in 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
```

Two variants of virtual environment can be started now.
Basic topology.

```
cd ∼/HelenOS/topology
./basic-topology.sh
```

Triangle topology.

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

Both are using image of HelenOS with installed BIRD created with previous steps.

NOTE: When work on this theses was finishing, there was still a bug in HelenOS, that caused all coastline builds fail with compile errors for ia32 architecture. There was a workaround fix for this bug. Dynamic linking needed to be disabled before BIRD installation, as described in
http://www.helenos.org/ticket/669.