Implementing a flexible network stack

Lasse Bang Dalegaard

Kongens Lyngby 2013
IMM-B.Sc.-2013-0028

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Sc.-2013-0028

Summary (English)

The diverse needs of modern networking technology have spawned flexible, high-performance packet processing engines like the network stacks in Linux and FreeBSD or the Click Modular Router. All of these provide high performance infrastructures that can be used to create complex networked systems. The large footprint of these systems however limit their applicability to relatively capable systems.

In this project, we design and implement Trokis, a minimal set of components that together can be used to create any network stack, ranging in scope from small embedded systems to large general purpose systems, like servers and work stations. We also implement several protocols on top of our framework, to showcase the modularity of our approach.

Summary (Danish)

De forskelligartede krav i moderne netværksteknologi har affødt højt-ydende systemer som netværksstakkene i Linux og FreeBSD, og den modulære routing platform Click. Disse udgør hver for sig en platform for skabelse af komplekse netværkssysstem, men deres størrelse gør dem uegnede til systemer med begrænsede resourcer, f.eks. indlejrede systemer.

I dette projekt har vi designet og implementeret Trokis, et minimalt sæt komponenter der tilsammen kan bruges til at skabe ekstremt fleksible netværksstakke, både til indlejrede systemer og til systemer med brede formål, f.eks. servere eller arbejdsstationer. For at vise modulariteten i vores metode designer og udvikler vi desuden et sæt protokoller oven på Trokis.

Preface

This results was prepared at the department of Informatics and Mathematical Modeling at the Technical University of Denmark, in partial fulfillment of the requirements of the degree of Bachelor of Science in Electrical Engineering.

This projects focuses on topics in computer science, some experience with object oriented programming, along with basic knowledge of networking systems, is assumed on the part of the reader.

2013-07-31 Lyngby

Lasse Bang Dalegaard

Acknowledgements

I would like to thank my supervisor Sven Karlsson for input during the project.

Contents

Chapter 1

Introduction

In recent years, the internet has seen extremely rapid growth, in terms of number of connected devices. One paper by Cisco, puts the total number of connected devices at more than 12 billion[1]. Each of these devices runs an entire protocol stack, allowing it to communicate with the outside world. Traditionally, protocol stacks have been built directly in to the operating system kernels, allowing relatively high performance. The stacks are however very advanced, internally tightly coupled, and allow limited modularity. In some cases, a general purpose stack, like the one implemented in Linux, is simply too heavy weight for a small microprocessor. Recent research in the field has yielded systems like Click[2] and Lightweight ANA[3]. These systems are extremely flexible, allowing high performance and great modularity, but are also quite heavy weight for smaller systems.

The goal of this project is to implement a lightweight, flexible userspace network protocol stack that can be adapted to a specific application with little or no change to individual components. The stack should be flexible enough to allow it to run completely in user-mode on any given operating system, ranging from high performance servers to embedded device. We emphasize that the stack should run in user mode, as this allows us to avoid many of the problems that plague kernel code and therefore most modern network stacks. It also gives us the opportunity to exploit modern operating system features like access to virtual memory and a complete standard library implementation along with the

newest advances in programming language technology. This approach allows a highly modular protocol stack architecture.

To show the flexibility gained with our approach, we show a relatively simple implementation of the IPv4 and IPv6 protocols and select transport layer protocols. The goal of this implementation is not to create a complete implementation of either of these internetworking protocols, but instead to focus on how the different components are connected, and what common infrastructure is required for them to work together. We will also show how this flexible scheme can be tied together with modern multi-queue enabled networking devices.

Our stack is implemented in C++ using the newest features from C++11. C++ was picked because it is a systems programming language that focuses on high performance and providing excellent abstractions for integration with low level systems. It is often supported on embedded and general purpose platforms alike and widely used.

During the course of this project, we have assumed a x86_64/AMD64 as the processor architecture. When running on embedded platforms, this will of course not be a perfect approximation in any way. While our stack tries to maintain portability where possible, several (mainly performance related) aspects will always be specific to a given processor.

We've chosen to name our stack Trokis, a loose transliteration of the Ancient Greek word τρόχις meaning courier or messenger[4].

Chapter 2

Networks and protocols

In this chapter we analyze the requirements of a modern ethernet based network stack.

2.1 Prevailing technologies

The International Telecommunication Union estimates that 2.7 billion people will be using the internet in 2013[5], corresponding to about 39% of the world population. Underlying the global Internet is the TCP/IP protocol suite. The IP protocol, the basis of the TCP/IP protocol suite, allows global addressing and is suitable for internet-scale applications. It is able to run over any type of packet switched network. The most widely deployed version of the IP protocol, IPv4, uses a 32-bit address as a global identifier, while a newer version, IPv6, uses 128-bit addressing with a new allocation scheme in order to ensure better scalability for many years to come[6].

While IP is able to run over any packet switched network, at end hosts(ie. servers, desktops and laptops) it is often transported over either wired ethernet(IEEE 802.3), Wi-Fi(IEEE 802.11) or WiMAX technologies(IEEE 802.16). All of these technologies are IEEE 802® standards and therefore follow the

IEEE 802® model. A protocol stack implementing support for all three would therefore be able to interoperate with a large number of available devices.

Cellular systems generally do not use the IEEE 802® family of standards for transporting IP, instead utilizing technologies like SNDCP(Sub Network Dependent Convergence Protocol) for GRPS(General Radio Packet Service, part of the Global System for Mobile Communications, GSM)[7] or PDCP(Packet Data Convergence Protocol) for UMTS(Universal Mobile Telecommunications System, a third generation(3G) cellular technology)[8].

2.2 Packet network fundamentals

When transporting data over a communication channel, two modes of communication are generally applicable: packet-switched mode and circuit-switched mode. In a circuit-switched system, a dedicated communication channel is established between nodes, thus the claimed resources are dedicated to this channel alone. This guarantees the communication channel all the allocated bandwidth, without the possibility of others interfering. This scheme is used in eg. the Public Switched Telephone Network(PSTN)[9]. In the PSTN system, channels were originally set up physically by mechanical equipment moving wires around, but in modern times this is handled digitally by setting up a 64 kbps digital virtual circuit, typically over Synchronous Digital Hierarchy(SDH)[9].

In contrast to circuit switched networks are packet switched networks. In a packet switched network, data streams are split into multiple packets which are then transported over the network as individual units, and then processed at the end node, where they are combined or interpreted in some way. Packets can be either fixed-size(ie. cell-based, eg. Asynchronous Transfer Mode(ATM)) or variable-size(eg. all IEEE 802® standards), depending on the underlying technology used. Further more, packet mode can either be connectionless(eg. Ethernet) or connection-oriented(eg. Frame Relay)[9]. The connection-oriented mode is also known as virtual-circuit switching.

2.3 Protocol stack architecture

Layering is a heavily used architectural viewpoint in the networking field. Here we will discuss some of the prevailing architectural viewpoints and how the previously discussed prevailing technologies fit in this picture.

2.3.1 Concepts of layering

It is generally recognized that a large part of the success of the internet is to be found in the layered architecture popularized by especially the TCP/IP suite[10]. The idea of layering is to introduce a framework that describes how different protocols work together in order to provide a specific functionality for the application running on top of them. The following are examples of layer functionalities, taken from the IETF model, the model underlying the TCP/IP suite[11]:

Error Control This functionality provides reliability for the communication protocol. An example could be the retransmission scheme of TCP.
Flow Control Control of channel data rate, for example to avoid congestion.
Fragmentation Breaks large chunks of data into smaller pieces. Examples include IP fragmentation and TCP segmentation.
Multiplexing Several higher level connections multiplexed over a single lower level connection.
Connection Setup Peer hand-shake, for example TCP three-way handshake or SCTP four-way handshake.
Addressing/Naming Management of host and entity identifiers.

Any arbitrary functionality besides the above could of course be provided by a layer. For example, IPsec[12] provides authentication and/or confidentiality as well as the normal functionality of the IP protocol.

The idea in layering is that a protocol at a given layer will only provide services to protocols from higher layers, and will only use services from lower layers. The benefit of this approach is clear: it allows us to "stack" protocols on top of each other, combining the functionalities from several protocols as required. An example is the use of a TCP connection over IP over ethernet. While there are more subtleties to this example, we can think of the "resulting protocol" as having the properties of all these protocols together:

TCP Provides error control, flow control, fragmentation and connection setup
IP Provides routing along with global addressing in the form of IP addresses.
Ethernet Provides local transport.

The resulting protocol will thus be reliable and be able to run over the global internet. This protocol would be great for web and mail traffic, since these require guaranteed delivery of data to function correctly. On the other hand, if we were to write a voice over IP application, we would not need the retransmission(error control) provided by TCP. Since the protocol architecture is layered, we can simply replace the TCP layer at the top with for example a UDP layer,

or run directly on top of IP. The resulting protocol stack would retain the global addressing and local delivery properties of the former protocol stack, but would not include error control, flow control, connection setup, etc. The layered architecture thus provides great advantages for real world networking systems.

This protocol layering is present in (at least) two forms in network systems: as implementation-specific software layers and as precisely defined protocol specifications. This distinction is important. When working between hosts across a network, protocols need to be exactly specified, so they can be encoded and decoded correctly. Failure to specify a protocol exactly could result in eg. endian issues or seemingly invalid packets. Inside a host however, the software managing the protocol layers is implementation defined, and does not specifically need to be layered at all, eg. the OSI model is heavily focused on layering in all parts of the architecture whereas the TCP/IP model in several cases discourages layering in the implementation. While one is "forced" to use the layering for the actual protocols, the software side could potentially be optimized in several ways if some other architecture but layering is used.

2.3.2 OSI layering model

The OSI model[13] is a generic layering model diving the protocol stack into seven different layers as shown in Table 2.1. Note that the model does not specify any layers above 7 or below 1.

#	Name	Functionality
7	Application	Provides services to the application
6	Presentation	Provides conversion between layer 7 and 5, eg. encryption
5	Session	Manages sessions between applications
4	Transport	Provides end-to-end connections, reliability and flow control
3	Network	Provides logical addressing and routing
2	Data Link	Provides physical addressing
1	Physical	Provides transmission over physical medium

Table 2.1: The seven layers of the OSI model

Within each of the seven OSI model layers are a number of entities, each of which is allowed to use the services of entities in the layer immediately below it. It is also allowed to interact with other entities at the same level, either directly or through a proxy in a lower layer. Every layer has a relatively fixed set of responsibilities. Communication between layers is performed through a conceptual port construct known as a Service Access Points(SAP).

The model also defines the terms Service Data Unit(SDU) and Protocol Data Unit(PDU), and one of each is defined for every layer. The PDU of layer N represents data that has been encapsulated by the layer N. Similarly, the SDU of a layer N represents the PDU of the layer N+1. When moving down the layers, the PDU is passed to the layer below, converting the PDU to a SDU. The layer entity hereafter performs some form of encapsulation, typically inserting headers and/or footers, resulting in a new PDU. The reverse process happens when moving up through the layers.
This model of encapsulation and decapsulation matches the actual protocol behavior used in real protocols.

The OSI model provides a good conceptual model of network protocol layering, and is useful for describing many systems, but it breaks down with many real world protocols. For example, the TCP protocol delivers end-to-end connections via ports, but also manages inter-application sessions. This puts it at both layer 4 and 5 of the OSI model, but if strictly following the OSI model, this is not allowed. Another example is the ARP protocol(and the IPv6-equivalent, Neighbor Discovery Protocol) for translating logical IPv4 addresses into physical link-layer(typically MAC) addresses. This means the ARP protocol performs part of both OSI layers 2 and 3.
Furthermore, some concerns, such as security, are allowed to affect more than one layer, again breaking the strict layering.

Despite these shortcomings the OSI model is heavily used as a guidance tool when constructing networking systems in the real world.

2.3.3 TCP/IP model

The TCP/IP model is the underlying model for the Internet Protocol Suite[14]. Unlike the OSI model, which is a generic model of networking systems, the TCP/IP model can be considered an implementation of the OSI model. There are however major differences in the approach taken between the two models. Whereas the OSI model focuses heavily on layering, the TCP/IP model favors architectural principles to layering[15]. The TCP/IP model does specify four layers but, unlike the OSI model, these are not numbered, only named. The layers can be approximately related to the OSI model as shown in Table 2.2 As seen in Table 2.2, the TCP/IP model does not concern the physical layer, and several of the upper layers are combined. The table is only approximate however, as eg. TCP provides several parts of both the Session and Transport OSI layers. Instead of focusing on layering, the TCP/IP model focuses on a few architectural design goals, especially simplicity, scalability, robustness and running code[15][14][11].

Table 2.2: Approximate relation between OSI and TCP/IP layers

TCP/IP layer	OSI layer #	OSI layer name
Application	7	Application
	6	Presentation
	5	Session
Transport	4	Transport
Internet	3	Network
Link	2	Data Link
-	1	Physical

The TCP/IP suite specifies that all protocols must use IP[16] as their Network layer protocol, in order to ensure interoperability between systems. The TCP/IP suite also favors a link-layer agnostic approach[14]: the TCP/IP suite must be able to run over any link-layer protocol, in order to decouple the two and allowing independent development and evolution at either layer. Therefore, the TCP/IP suite itself specifies no link-layer but only a set of requirements for the link layer protocols supporting it[16]. Some specifications for transporting IP over various link-layer protocols do however exist, notably RFC 894(IP over Ethernet II networks)[17] and RFC 1042[18](IP over 802 networks).

2.3.4 IEEE 802® layers

Ethernet and WiFi are both defined by IEEE 802® standards, defined respectively by the working groups 802.3 and 802.11. All IEEE 802® standards share the same layering Architecture[19], matching the OSI layering model with a Physical and a Data Link layer. The Physical layer of course defines the signaling and physical characteristics that devices must follow. This is technology-specific. The IEEE 802® standards split the Data Link layer into two sublayers, namely the lower-level Medium Access Control(MAC) sublayer and the higher level Logical Link Control(LLC) sublayer, connected via service access points. Optionally, some IEEE 802® LAN standards provide an additional sublayer, the Ethernet sublayer, operating on top of the MAC layer in the same way as the LLC sublayer. The MAC sublayer is technology-specific, but the LLC and Ethernet sublayers are shared among all IEEE 802® standards.

The MAC layer is a purely frame-based, connectionless layer, responsible for data transfer between hosts on the local network. Frame-level functions are all handled at the MAC layer, including frame delimiting and recognition, physi-

cal addressing via a 48-bit MAC-address, error protection(eg. checksumming), and of course payload transfer. The MAC layer can also implement other features beyond this, for example flow control(eg. PAUSE frames in 802.3x) or encryption(eg. 802.11i, Wi-Fi Protected Access 2). The MAC layer is typically implemented as software in the host protocol stack, although it can of course make use of hardware offloads if the underlying network interface device provides them.
As noted, the MAC layer is responsible for performing encapsulation of LLC and optionally Ethernet frames. The choice between either of these is defined by the network standard in question, although most IEEE 802® networks always use LLC, with 802.3(ie. wired ethernet) being the exception, typically using the Ethernet sublayer directly[17].

The Ethernet sublayer is a relatively simple encapsulation, consisting of a two octet protocol number(known as the ethertype), followed by packet data. An ethertype value in the range 0-1500(both inclusive)[20] indicates the the frame is using LLC encapsulation, and in this case the ethertype specifies the length of the payload of the frame(ie. the LLC payload length). An ethertype above or equal to 0x0600(1536) is used directly for dispatching to a higher level protocol. The list of ethertypes is maintained by IANA[21]. The Ethernet sublayer thus provides unacknowledged connectionless packet transmission.

The LLC sublayer is responsible for providing several services on top of the MAC layer[22]. LLC provides three different service types on top of MAC, namely:

Type 1 Unacknowledged Connectionless-mode
Type 2 Connection-mode
Type 3 Acknowledged Connectionless-mode

Only Type 1 is required for standards compliance[22] and this type is used in the vast majority of cases, as session handling is typically done at the transport layer, for example in TCP. Furthermore, RFC 1042 recommends the use of Type 1 LLC when running IP and ARP over LLC[18].

The LLC layer also provides multiplexing capabilities by way of the DSAP(Destination Service Access Point) and SSAP(Source Service Access Point). The DSAP indicates the upper layer SAP(ie. protocol) that should receive the encapsulated information contained in the LLC frame. The SSAP indicates the SAP(ie. protocol) of the sending protocol. Both DSAP and LSAP are 8-bit fields, and when both have values of either 0xAA or 0xAB, the LLC frame is Subnetwork Access Protocol(SNAP) packet[19]. In this case, the initial five octets of the LLC frame make up the SNAP header, and the first three octets of this header constitutes an OUI, while the final two octets are the protocol number, vendor(OUI)-specific number. In the case where the OUI value is all-zeros, the inner frame of the

LLC/SNAP packet is an Ethernet sublayer packet. The SNAP protocol number in this case directly matches the Ethernet sublayer ethertype value.

2.3.5 End-to-end principle

The end-to-end principle specifies that some piece of functionality should be implemented in the end-nodes of a communication path, provided that the implementation can be done completely and correctly[23].

The end-to-end principle is extremely interesting in the context of a protocol stack, because of it's implications in the placement of various components. For example, many modern operating system kernels provide the TCP and UDP protocols in the kernel[24, 25]. Following the end-to-end principle however, it could be argued that there is no reason for these protocols to live in the kernel. The end-node would in this case be the user-space application using the TCP or UDP protocol. There is is no obvious reason preventing the user-space application from implementing (at least most of) the required functionality "completely and correctly", so it follows by the end-to-end principle that TCP and UDP should be provided in the user space application. Of course, there is a small extra detail to this, namely the fact that the protocol stack will still need to be able to perform some inspection of TCP and UDP, in order to forward the correct ports to the correct user-space application. This is however only a small part of the functionality provided by at least TCP, and this small part of the functionality could be provided by the kernel, and the rest could(eg. connection setup, flow control, etc.) could be provided in the user-space application.

Applying end-to-end principle can improve performance in several internet applications. In the example of TCP and UDP, the kernel maintains a buffer of received data(or packets in the case of UDP) which the application must then retrieve from the kernel. The buffer typically also has a fixed size. This introduces a feedback loop where the kernel receives data, notifies the application to pull out the data, whereafter the kernel can again receive new data.

Another example of performance improvements by moving functionality to the end-hosts is seen in IPv6, where fragmentation is never performed by intermediate routers, unlike in IPv4 where routers are supposed to fragment packets larger than the underlying media can support. IPv6 instead mandates that all implementations must support packets of at least 1280 bytes[26], and that the end-hosts must perform Path MTU Discovery[27] in order to determine if the path between them can support a larger MTU.

It is important to remember that the end-to-end principle is just that, a prin-

ciple. It is not a rule that specifies exactly how things must be done, but it provides a good guideline when building network systems.

2.4 Modern network card architecture

Modern ethernet networks can achieve speeds in excess of 10 Gbps. The minimum length of an 802.3 ethernet frame is 64 octets, however the physical layer adds a preamble and interframe gap, resulting in a minimum allowed frame on a wired 10 Gbps ethernet connection of 84 octets. This results in a maximum frame rate of 14.88 Mpps(millions of packets per second), which the sender and receive must be able to process and possibly sustain for extended periods. The total time between each arriving frame is thus only 67.2 ns. This corresponds to only 201.6 cycles of a 3 GHz CPU. The CPU may need to perform route lookups, firewall lookups, consult a connection tracking system, pass the data to an application(which must interpret the data) and more, processing every frame in 200 cycles is not very realistic. Modern systems mitigate this by using multiple parallel processing cores, each working independently. As much of the packet processing is (close to) embarrassingly parallel, the task can be efficiently be split between CPU, allowing almost linear speedup. Separate solutions are implemented to accomplish this in both egress and ingress directions.
Furthermore, modern network interface cards make extensive use of DMA in order to perform as much copying as possible without host CPU intervention. It should be noted that different cards support different addressing widths, potentially limiting the amount of memory available for packet buffering. This should be taken into account in the design.

Most modern network interfaces further offer many different offloading functionalities, allowing the host CPU to use less time on packet processing. Examples of advanced offloading include header splitting(ie. splitting the received frame in header and payload), stateless packet parsing(parsing VLANs, protocols, MPLS tags, etc.) and various stateful offloads eg. TCP offloading.

2.4.1 Ingress path

The ingress path of a modern network interface card[28] implements multiple independent receive queues that are used for frame reception. The number of queues available varies, but for the Intel 82599-based cards used for this project, 128 seperate queues are available. Frames are generally moved directly to system memory using DMA, and in some cases the CPU caches are also primed with

the data, allowing lower latency.

Upon reception of a frame, the network interface card will perform various processing steps on the frame, finally placing picking a queue that must receive the packet.

The queues are implemented as ring buffers, and as such they are of a fixed capacity and of FIFO-ordering. When the host is ready to receive a packet, it inserts a so called receive descriptor into the ring. This descriptor includes flags and addresses the network interface card will use when writing the packet to system memory. Upon reception, network card will fetch the next descriptor in the ring, and if it encounters a receive descriptor it will use the data within to push the received packet directly to main memory using DMA. The network interface card then performs a so called descriptor write back, updating the receive descriptor with additional data. The packet has now been moved to main memory and is ready for processing by the host. The host may either busy wait(or poll) on the ring buffer, or may receive interrupts from the network interface. Which strategy is chosen depends on a number of factors, as interrupts and busy waiting have different trade offs. Most modern network stacks allow both modes and adaptively switch between them when the rate of interrupts generated gets too great.

Since there are multiple queues active, each one may be assigned to a separate processor, using a load-balancing scheme generally termed Receive-Side Scaling(RSS)[28]. In this mode, the network interface card generates a hash of some of the headers of the receive packet. Generally a "level 4" hash is used, referring to the OSI transport level. In such a hash, level 4 and usually also level 3 data is used to generate a hash for the packet. For UDP/IP and TCP/IP packets, port numbers along with IP source and destination are normally used to generate the hash. The resulting hash is then used to select a receive queue for reception of the packet. This is done to ensure that a single flow is always steered through the same queue, in order to ensure in-order delivery. While some protocols(eg. TCP) can cope with out of order packets, performance is reduced in these cases. While this load-balancing scheme does not ensure a perfect split of traffic between receiving CPUs, it usually results in a relatively fair balancing.

Several other queue-mapping strategies besides RSS are possible. The Intel 82599 for example supports using 8 queues(each with a different frame priority) per CPU, with up to 16 CPUs(128 queues total), and various modes for Data Center Bridging, special packets(eg. TCP SYN packets) and more.

Assigning a queue to a single processor has the added benefit that locking at the queue level is no longer required between host CPUs, as only a single CPU will ever work with the queue. This removes all contention overhead.

2.4.2 Egress path

The egress path a modern network interface card is many ways conceptually similar to the ingress path. Frame transmission is done using multiple separate, independent, transmit queues. As in the receive case, transmit queues are implemented as ring buffers, with a bounded number of slots. Each slot can be filled with (part of) a frame to be transmitted. The network interface card will fetch as many descriptors(and the data they point to) as required to assemble an entire frame. Upon completion of the frame, it is transmitted to the network.

One the ingress path, Receive Side Scaling is used to load-balance between queues. On the egress path, a "reversed" scheme is used. The host is responsible for programming the network interface card, telling it what queues to prefer when sending. Several options are possible here, for example plain and simple round-robin, or more advanced schemes based weighted eg. round-robin or strict priority. When sending, the network interface card will pull descriptors out of the transmit rings in the programmed order, thus achieving load balancing between sending CPU cores.

Once a packet has been transmitted, the network interface can use one of several mechanisms to signal completion, eg. interrupts, writing a new ring buffer head pointer, or updating the descriptors it processed.

As was the case with mapping a single processor to each ingress queue, mapping a single processor to an egress queue removes locking requirements between cores, removing contention overhead.

Chapter 3

A basic framework for protocol implementations

In this section, we will discuss the various components required to support high performance packet processing in a network stack. The focus here is not on specific protocols, but rather on a general model of packet processing. We will start out by exploring how packet payloads can be managed and modified, and will then continue on to how processing elements can be defined to make use of this infrastructure. We will also look at how concurrency can be handled, as it poses a non-trivial problem but is vital for achieveing high performance on modern multi-core machines.

3.1 Prototype implementation basics

The Trokis prototype framework is implemented as a pure header library in C++11. A pure header library was chosen because it is very easy to work with for the programmer, requiring no special linker options or other compile time options. This does increase compiliation times considerable as everything is pulled together in a single translation unit, forcing a complete recompile in all cases. The performance of the compiled program, when assuming that link-time optimizations are available, should be similar in both cases.

3.2 Packet payload management

In order to process packets, we need a way to manage packet payloads. Since multiple different protocols will need to interoperate, we need a model that is generic enough to support all common patterns used in communication protocols. We also need the interface to be portable, so that the network stack can be used on big and little endian machines with no modifications.

At the lowest level, we can simply think of a packet payload as a series of bytes representing some data. Most protocols today are based on either prepending headers or appending footers to data that they encapsulate. Examples include all protocols in the IP suites(including IPv4, IPv6, TCP, UDP and ICMP), along with the IEEE 802.3 ethernet protocol. Our abstraction must efficiently support these functionalities.

For this purpose, we define `packet_data`. The `packet_data` interface encapsulates a sequence of bytes contained between two points known as Head and Tail. The Head point signifies the begining of a packet payload, while the Tail point signifies the end of the payload. The area between these two points is know as the Payload area. The user can grow the payload area infinitely in both directions, and can of course also shrink the payload area to a minimum size of zero bytes(no payload). Resizing can occur indepedently at either end. All positions before the Head and after the Tail pointers are undefined, but may be present in memory. A visualization of the abstraction can be seen in Figure 3.1. It is important to note that the Head and Tail points are not pointers, but are simply known indicies for the programmer. The Head point by definition always has the index zero, while the Tail pointer has the index $length - 1$.

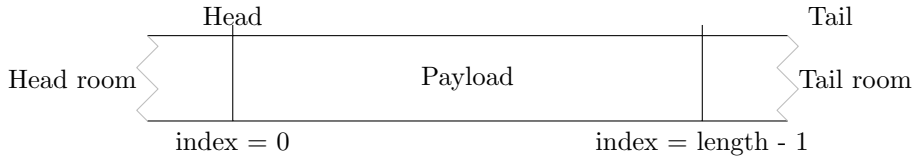

Figure 3.1: `packet_data` abstraction. The Head and Tail pointers can be moved independently, growing or shrinking the payload area in the given direction.

The Head and Tail points can be moved using the functions `push`, `pull`, `put`, and `trim`. The `push` and `pull` functions move the head point respectively backwards and forwards, ie. extends and shrinks the payload area. Likewise, the `put` and `trim` functions move the Tail point forwards and backwards, ie. extends and shrinks the payload area. This interface is very intuitive to use, and eg. allows

the programmer of an IPv4 processing block to call `push(20)`, prepending a 20-byte header to the current payload. It should be noted that when expanding the payload area, the values of the bytes in the expanding area are undefined, and it is thus up to the programmer to fill them in. Zeroing out these bytes would waste time, as the programmer will most likely fill them in with new values in any case.

The actual implementation backing `packet_data` is not specified by the interface, but it is specified that it must consist of a limited number of byte arrays. The interface provides an interface for retrieveing the list of byte arrays backing the `packet_data`, but these byte arrays should not be modified directly. This interface is provided for the few cases that need raw access to the underlying data. This would mainly be the case when moving data outside the infrastructure we provide.

The advantage of this data management scheme is that is completely frees the programmer from having to worry about the underlying memory structure of the packet data, and likewise it gives complete freedom for implementing the backing data storage for the payload. For example, the implementation could reserve extra space at the head of the packet, allowing protocol headers to be appended without having to move or copy the data in the payload. Allowing fragmented data buffers also becomes trivial, as no part of the upper level interface will have to change. This can potentially have huge performance benefits in some situations, because is removes or reduces the need for copying the entire backing buffer when segments are added.

On top of `packet_data` is a layer of helper functions for getting and setting bytes in the packet payload, along with a function for performing boundary checks. Letting the programmer explicitly handle boundary checks allows better performance because boundaries can be verified just once and then simply assumed to be correct from there on out. The byte getting and setting helpers are implemented in C++ as a very small template library operating on pure iterators. This allows the compiler maximum freedom to optimize and inline the code when it deems this to be more efficient. Wrappers around this library is also present directly on the `packet_data` interface. This does have the drawback that the algorithms(the helpers) and the data(the payload contained in `packet_data`) is not entirely decoupled, but it seems to feel more natural for the programmer, especially for one coming from an object oriented background, to have the functions as members of the `packet_data` structure.

The helpers allow type safe reading and translation of integers in the byte stream to host endianness integers. This is important for portably, as it allows processing blocks to be written in portable code rather than having to assume some byte order, or having to perform checks in the code. This does come at a po-

tential performance cost, but it was felt that this cost was generally very low and could easily be justified by the increased productivity gains. Most of the performance implications can be hidden or entirely removed using processor specific instructions for performing byte swapping. The current version also simply reads single bytes, rather than reading the largest possible type(for example, reading 32 bit numbers as one 4-byte word instead of as four 1-byte words). This version is however simpler to code, and should be easy to change if it proves to be a bottleneck.

An early prototype of the helpers used a much more elaborate scheme, where entire on-wire structures could be described at bit resolution and decoded. This was based on heavy C++ meta template programming and would potentially have allowed fully type safe decoding of structures, rather than the non-portable use of bit fields and functions for translating between byte orders as is often seen in low-level C code. This approach however had some even larger performance implications, as there was no obvious way for the compiler to perform optimizations across several reads from the structure. Allowing the programmer to perform these optimizations manually would have removed the type safety features, and essentially nullified the entire concept. After some deliberation, it was decided that the before mentioned approach of providing simple integer helpers would be a better fit for our implementation, as it does not hide when reads or writes are made and allows the programmer to perform these optimizations themselves if they so choose. This also forces the programmer to think about when packet data is accessed, which we consider to be a useful effect.

Each live packet in the system is managed by an instance of the `packet` class. The packet class is really just at POD(plain-old-data)-type encapsulating a pointer to the assigned `packet_data` object, as well as a few very basic pieces of meta data. The meta data included is currently limited to:

worker The ID of the worker that is currently assigned to this packet. This field is used mainly for implementing concurrency and is explained further in Section 3.4.

dev The ID of the "current" device of the packet. The meaning of this field is dependent on which direction the packet is travelling through the stack. When a packet arrives on an interface, the dev field is filled with the interface ID of the device that received the packet. When the packet is getting transmitted, the dev field is filled with the device ID of the device that should transmit the packet. This field is in the `packet` type because it is so central to the function of the network stack, and the notion of the networking device(or interface) generally transcends the layers of the stack.

More fields could potentially be added to the `packet` type, but in most cases pa-

rameters should be moved between blocks using function call parameters rather than packet state. This improves flexibility because blocks only need very basic knowledge of the `packet` type. Generally, only fields that transcend all layers of the protocol stack should be added to the `packet` type. The device ID is an example of such a field. For example, the IPv6 protocol generates link-local addresses that are meaningless without knowing the physical interface, but the IPv6 addresses will never be important to the IPv4 module. In this example, IPv6 addresses should be passed via function call arguments, while the device could be transported via the `packet` type. Adding a field to the `packet` type reduces flexibility, but does give a potential performance increase as it potentially frees up a register when performing calls between processing blocks.

Note that there are no pointers to block-specific datastructures in the `packet` type. This is again to maximize flexibility. While moving pointers directly to the `packet` type would result in better performance when looking up configuration data, for example being able to immediately dereference the worker or dev fields as pointers, this introduces tighter coupling between the components of the stack. Instead, we force processing blocks to implement their own indirection tables. Several blocks that are designed to work together could of course share their underlying data structures in this regard, but our design tries to encourage each block to be independent if possible.

In order to allow high performance moving of packets between different parts of the stack, we always pass pointers to encapsulate `packet` objects. We simply use standard C++11 `std::unqiue_ptr`, because this ensures that the object is always released when the pointer goes out of scope. Using rvalue-references we can safely move the packet data around the stack, ensuring that a packet is never has multiple owners. This also means that we cannot simply copy packets, but there seems to be no need for this in most cases. Sniffers and a "tee" processing block would require the ability to copy a packet, but these could simply spawn a new packet of the correct length and copy the data from the original packet to the new packet. The extra safety gained from not sharing data outweighs this drawback.

3.3 Processing elements

An important aspect of a protocol stack is the ability to compose various protocols to form a complete stack. For example, an application needs to be able to use TCP, which needs to be able to run on top of IPv4, which in turn needs to be able to run on top of Ethernet. The interfaces between each of these components varies, and in some cases more steps are needed in between them,

in order for correct protocol operation.

Instead of defining a common interface that all protocols at a given layer must use to communicate with surrounding layers, we instead define a set of rules on how a protocol should be implemented in order for other protocols to interact with it.

At the heart of our design are protocols, which we encapsulate in generic units called "Proessing elements" or "Processing blocks". Each block is a C++ class, encapsulating all state information that the block needs in order to perform its task. We do not use dynamic dispatch(virtual function calls) with these classes, but instead treat them more as namespaces that we can spawn instances of. This has the benefit the functions and state are contained in a single unit. This is ideal for our project, as we want processing blocks to be maximally interchangable with as little modification to processing blocks as possible.

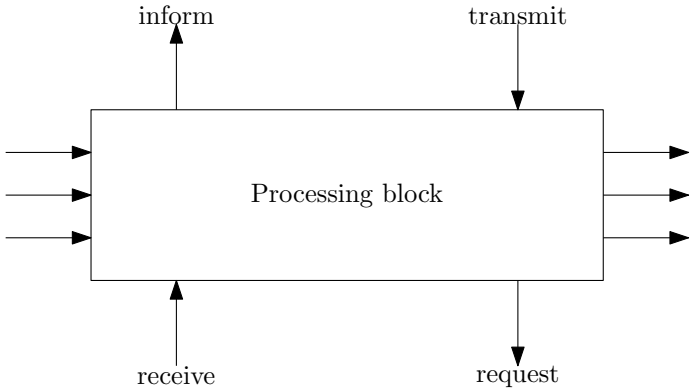

Figure 3.2: Generic processing block. Pointers to the block are function calls(entry points), pointers away are hooks(function pointers). No functions or hooks are required, the named ones are simply those that will be commonly defined.

Each block has a number of input functions and a number of hooks, as illustrated in Figure 3.2. The input functions are the interface that other blocks(and external components) use when they need to use the services of a given processing block, or when they need to send configuration messages to the block. Each input has a clearly defined function signature, but each block can define exactly the functions needed for operation. The semantics of a given member function is also left up to the implementation of the block.

Perhaps more interestingly, blocks can also have a number of hooks. Hooks are

just function pointers, implemented in our solution as C++11 `std::function` objects. Hooks are set during setup of the stack, and never changed after this. This means that we do not have to surround the function objects with locking, atomic pointers, etc.

An object can have any number of hooks, and will typically also have a number of setters for setting each of these hooks. Blocks will also typically provide a default implementation of a given hook. Since hooks are not plain pointers, we can bind not only plain functions but also member functions(both virtual and not), function objects(functors), and anonymous functions(lambdas/closures). We also get a compile time guarantee that the function that we are assigning to a hook is compatible with the hook signature. An alternative to this approach would be to simply use the object oriented features of C++, namely (multiple) inheritance and virtual function calls(runtime polymorphism). There are numerous reasons why we picked function pointers over virtual function calls:

- Functions, not objects, are the basic building blocks of data processing. Functions perform actions, while objects simply encapsulate functions and data.
- Function pointers can be potentially as efficient as virtual calls(since virtual calls are typically implemented as a table of function pointers).
- In some cases, we may wish to map the hook of one processing block to a method on another processing block, in which case the function call signature of the hook and method need to match. Were we using virtual functions, we would need to implement a class with a single method to perform this mapping. Using function pointers, the mapping could be done using an `std::bind` expression or an anonymous lambda function, directly at the location where we set the hook. This makes it easier to see what is going on.
- The function pointers can point to both member and non-member functions, allowing easier interfacing with C code if so desired.

There are some drawbacks of this approach. Performance wise, both virtual function calls and function pointers suffer compared to direct function calls. This is because direct function calls do not involve any kind of runtime lookup in order to get the address that must be called to, resulting in both fewer instructions and better branch prediction. Furthermore, direct function calls allow the compiler to perform inlining, entirely eliminating both parameter passing and call setup costs. Direct calls are however simply unacceptable in our system, because they by their very nature cannot be changed. A hybrid approach could most likely be created, where the type of each hook was passed as a C++ template argument to a processing block. This would allow the compiler to treat calls as direct and thus allow inlining, at the cost of vastly increased compilation times and possible code bloat resulting from the large amount of template instantiations. It is

doubtful that this would result in a net increase in performance, as the the extra code size could potentially lead to more instruction cache misses which would have to be contrasted with the already relatively low performance overhead of a virtual function call. Finally it should be noted that the code pointed to by most hooks will simply be too large to inline, and that the lambda functions or `std::bind` expressions that are used to adapt the call from hook to function will be partially inliniable(the call to the adaptor will be virtual but the call from the adaptor to the target will/can be direct). The relative overhead of performing a virtual call should therefore be very small, and unlikely a problem.

Relative to virtual function calls, `std::bind` expressions can quickly become hard to read, and current lambda functions in C++11 do not allow polymorphism, forcing us to type out complete types for all the hook parameters that we are mapping. This quickly becomes tedious, error prone and hard to read. With C++14, lambda functions will allow polymorphism, allowing types to be specified simply as "auto". We however feel that the advantages by far outweigh the negative implications of using virtual functions.

The processing blocks in our system are not designed to be exchanged at runtime. Once the system is started and initialized, the layout of the processing blocks should remain constant. This allows frees us from using atomics or locking when dereferencing hook function pointers.

In general, our approach does not force any specific interface for processing blocks, but many protocol blocks need ways to communicate with upper and lower layers, and for this reason we have selected a standard naming convention for these hooks and functions. For introducing data to a given processing block, we have two functions named follows:

Receive A lower layer calls Receive to indicate that it wishes to communicate data to upper layers. With the call it passes the pointer to a packet along with any required data to fulfil the interface specified by the receiving processing block.

Transmit A upper layer calls Transmit to indicate that it wishes to communicate data to lower layers. A pointer to a packet, along with any additional arguments, are supplied as parameters.

Furthermore we have two hooks, named loosely after the naming convention used in the IEEE 802® standards:

Request The Request hook is invoked when a processing block wishes to communicate with a lower layer protocol. The packet pointer along with any parameters required by the lower level interface are passed on.

Inform When a processing block wishes to communicate data to upper layers,

it invokes the Inform hook. Again, a packet pointer and additional data is passed.

Conceptually, we can think of the Inform hook of a layer N processing block being connected to the Receive member of a layer N+1 processing block, while the Request hook of a layer N processing block will be connected to the Receive member of a N-1 processing block. In practice, adapters may be introduced between the processing blocks, for example other processing blocks or plain functions.

Configuration of a processing block is done outside of the normal flow of the processing block framework. This allows each processing block to handle configuration in the most efficient way possible. For example, the configuration of a generic block that simply dispatches based on a parameter would be very different from the configuration of a block that handles the IPv4 protocol. By leaving configuration up to the processing block implementer, we give them the flexibility do select the best possible configuration scheme. This also allows us to perform complicated setup actions when required. For example, when an interface is registered we may need to set up IPv4 and IPv6 for the interface. Forcing a particular flow would decrease the applicability of our processing block components.

3.4 Handling concurrency

An important problem in scaling a network stack is how concurrency will be handled. Modern network interface cards support multiple receive queues, allowing packets to be steered directly to a specific CPU on the receive path and allowing each CPU to be allocated dedicated transmission queues on the transmit path. In order to take advantage of these, relevant parts of of our network stack must be thread-safe.

Packets in our network stack are always handled by only a single thread at any given time. This is enforced by the use of `std::unique_ptr`, as long as we do not pass references to other threads. This frees us from having to use locks or even atomics around `packet` objects and the `packet_data` subobjects, allowing maximum performance when manipulation these objects.

When implementing thread-safe code, there are two overarching paradigms to consider: blocking and non-blocking algorithms. Blocking algorithms using locking to provide mutual exclusion between threads, making sure that only a single thread is ever in the locked(critical) section of code. This has obvious conse-

quences for scalability, especially when scaling to large amounts of processors, because potentially hundreds of processors could be waiting for a single processor to leave the critical section. Blocking algorithms can generally not guarantee system wide progress, because the holder of the lock could be suspended indefinitely, for example by the operating system, waiting for an external resource, or by coding error(eg. an infinite loop). On the other hand are non-blocking algorithms. These algorithms typically work by using atomic CPU instructions, such as compare-and-swap, fetch-and-add or test-and-set in combination with full or partial memory barriers. These instructions ensure ordering between workers, allowing much better scalability than a lock-based solution. These are generally divided into wait-free and lock-free algorithms. Wait-free algorithms guarantee that all processors will make progress in a bounded number of steps, while lock-free algorithms only guarantee that the system as a whole will make progress.

While non-blocking algorithms guarantee progress and scalability, this says nothing about the actual performance of a given algorithm, and in general a blocking algorithm may be faster than a non-blocking algorithm. Furthermore, blocking algorithms are typically much easier to develop and reason about, and making them deadlock-safe is relatively easy, especially when using the C++11 lock library, utilizing RAII locks(automatically released at the end of a scope) and deadlock avoidance algorithms. Especially very short lock can potentially achieve very good performance, and the C++11 lock library includes `std::mutex` for performing this type of blocking. The performance of this mutex implementation depends on the operating system, but a modern operating system like Linux will typically provide a high performance mutex implementation. In Linux this is provided by the futex(Fast Userspace muTEX) construct in the Native POSIX Threading Library(NTPL). Futex performs as much as possible in userspace, only calling the kernel when the lock is contended. It is possible that a spinlock will give better performance, but this is hard to know ahead of time and will most likely depend on the load of the system, as higher loads will presumably lead to a higher chance of the current lock holder being preempted, stalling other threads. Measurements can show if a given lock is a bottleneck. Furthermore, reader-writer locking could be used to achieve parallelism between multiple readers. We have not used this scheme in our prototype(instead prefering optimistic concurrency control for these cases), but with the introduction of C++14 `std::shared_mutex` this functionality may be included in the C++14 standard library making it much more widely applicable.

Because of the difficulty of knowning ahead of time if a given lock will be contended, combined with the fact that blocking algorithms will often perform better in the uncontended case, we have decided to use optimistic lock-free algorithms when working with seldomly changed data, and blocking data structures as an initial implementation in most parts of our system.

In our network stack we allow for both blocking and non-blocking approaches to be taken, leaving it up to the implementor of a given processing block to decide which solution is best for the given case, for example using a lock-free structure for the routing table(heavily read, seldom updated) and combining blocking and non-blocking structures for IPv4 packet reassembly. The basic framework does not require any type of thread-safety, as the `packet` objects are only operated on by a single thread and the processing blocks are connected before packet processing starts.

3.5 Safe memory reclamation

Non-blocking data structures give rise to a unique problem: when data is removed from a data structure, there may be other threads traversing the same data. If the data is simply deleted once removed from the data structure, the other threads will have the data disappear without their knowledge, resulting in undefined behavior and crashes or worse yet, security problems. This problem is related to the so-called "ABA problem" which happens from the use of the compare-and-swap paradigm. If a thread reads the value A, and performs some calculations and checks that the value is unchanged(is still A), another thread can change the value from A to some other value("B") and back to A, without the first thread noticing. This is mainly a problem when using compare-and-swap with pointers, as it is possible for the object A to replaced by some other object B, deallocated and a new object reallocated at the same address(for performance reasons, modern memory allocators will try to cache memory blocks if possible). The result can be lost updates or worse(security issues, crashes, etc.).

Both of these problems can be eliminated if it can be guaranteed that data structures will never be deallocated while there are still threads traversing the data structures. The easiest way to ensure this is by using a garbage collector. While garbage collectors allow very easy management of memory, there are associated performs costs to using them. When using reference counting, memory is reclaimed exactly when it is no longer in use(thus reference counting is accurate), but there is a potentially high performance penalty resulting from contention for the atomic reference counters used in this scheme. An alternative solution is the use of a mark-and-sweep with eg. incremental or generational collection, like the Boehm garbage collector[29]. Garbage collection can provide very good performance, but there will always be a cost associated, especially in a language like C++ where only conservative collectors, like the Boehm collector, are really viable.

An alternative scheme is to simply defer data structure destruction long enough

for the data to no longer be in use. There are various ways to accomplish this that don't require holding any extra reference count, shadow stacks, hazard pinters, etc. but instead work "globally". Examples include Quiescent-State-Based Reclamation(QSBR), Read-Copy-Update(RCU) and Epoch-Based Reclamation(EBR)[30]. All of these work by in some way signaling that a given thread has reached a quiescent state, ie. a state where the thread was not inside a critical section, and therefore must have dropped the reference to the data. Memory can then be reclaimed by registering callbacks to be called once all threads have progressed at least one quiescent state, ensuring that all references have been dropped. The performance of all of these schemes is very good, RCU being used as one of the primary synchronization primitives in the Linux kernel. The biggest benefit of all of these is that there is no per-pointer overhead: all pointers are raw pointers, and can be deferenced as normal pointers, resulting in almost no performance penalty overall. RCU as used in the Linux kernel can take advantage of various kernel-only facilities, especially that it can disable preemption(in non-realtime kernels), disallow sleeping within the RCU critical section, and use the kernel processor preemption as quiescent state signalling. EBR and QSBR are more general than RCU, but overall work on similar principles.

All three of these are relatively complex, but we can create our own scheme based on some of the ideas. Rather than using quiescent states, we simply defer reclamation for a set amount of wall time, ten seconds in the current implementation. Once a piece of a data structure is ready for reclamation, we save the current time and enqueue the pointer onto a list of objects ready to be reclaimed. The list is a wait-free multiple producer-single consumer FIFO, allowing maximum performance for the producers(forwarding threads). A garbage reclaimation thread continuely pulls out a single object from the list and waits until the garbage object is older than the timeout, at which point the object is deleted and the reclaiming thread loops again. The garbage enqueue operation is wrapped in a non-copyable, movable smart pointer, `gc_ptr`, made possible by C++11 move semantics, allowing safe RAII-style management of the object lifetime. The smart pointer maps an atomic pointer, and thus there is no performance drop associated with managing the pointer. The pointer is non-coypable, to force single-owner semantics. This is required to stop double-free errors. An alternative would be to reference count the number of owners of an object, but this would result in extra overhead in cases that do not need shared ownership. Deletion has a small overhead, resulting from having to enqueue onto the reclamation list, but this is wait-free and only takes a few instructions, but the atomic exchange instruction used may of course cause a slowdown when many threads are enqueuing at once. This could be amortized by using thread-local deletion queues. In any case, the overhead of enqueuing onto the list should be much lower than the actual call to `delete`, resulting in an extra performance boost. Our solution retains the ability to use raw pointers.

Of course, this scheme will fail if a thread ever holds on to a deleted object for more than ten seconds, but given that the usage is network packet processing, this seems extremely easy to ensure. If a network packet is ever more than around a second old, interactivity will be completely destroyed and generally it is better to drop traffic this old. As an example, the CoDel queuing strategy drops traffic when it has resided in a queue for more than five milliseconds[31]. For this reason, it is very seldom that a packet processing step will need to hold on to a data structure for more than a few seconds, let alone ten. Simply using the fixed wall clock deferred time results in a vastly simpler implementation, that for almost all purposes in a network stack will be good enough.

Our memory reclamation scheme is not suited for extremely high garbage rates, because of the lag between enqueing and actual reclamation, but for most data structures we feel that the lag should pose very low overhead. On the other hand, our scheme will always reclaim the data after the threshold. This is a benefit over a scheme like RCU, which can potentially retain memory forever if a bug prevents even a single processor from reaching a quiescent state. Currently there is one major fault with our reclamation scheme, when reclaiming data structures with nested garbage collected pointers: when a garbage collected object is destroyed and its destructor called, new garbage collected objects can potentially be enqueued for reclamation. These will not be reclaimed before their own timeouts run out. This could potentially be a problem with deeply nested structures. The maximum reclamation time would in this case be $n \times T$ where n is the depth of the data structure being reclaimed and T is the garbage timeout threshold, with T equalling 10 seconds in our prototype. A solution could be to delete garbage directly when executing in the reclamation thread, but this is currently unimplemented in favor of simpler code. If this turns out to be a problem, a fix could be implemented.

Our safe memory reclamation scheme is of course optional for the processing blocks, and a given processing block may use any reclamation scheme demmed fit by the designer. This includes hazard pointers, pass the buck, proxy collection(very similar to our approach, but uses a reference count to a proxy object rather than a wall clock time based queue), any of the above mentioned quiescent-state-based algorithms, or some entirely different scheme.

Note that `packet` and `packet_data` objects do not need garbage collection, as they are only operated on by a single thread at a time, hence there is no risk of another thread having a reference to any of these structures when they are freed.

Chapter 4

Protocols

Based on the framework discussed previously, we will here discuss how IEEE 802.3, IPv4, and IPv6 can be implemented on top of the framework. We will also discuss how the network stack could be connected to external drivers and applications, as would be the case when used in a general purpose operating system.

4.1 Generic dispatch block

In several cases, for example in IEEE 802.3 and IP, a protocol produces a numeric value that indicates the type of encapsulated data. For these cases, we provide a generic block that receives a pointer to a packet along with the produced protocol number and remaining parameters and based on a lookup into a table calls the upper layer protocol. Our implementation uses the `std::map` type(typically a red-black tree) from the Standard Template Library as the backing container. Since the STL is used, we can easily switch the container if required, for example if it proves to be a bottleneck. We encapsulate each available handler in a `std::function` object, allowing us to store any function or member with the given function prototype, including lambdas and `std::bind` expressions. We also provide the possibility of setting a fallback handler that will be called if no registered handler with the given protocol number is found.

The generic dispatch block shows how C++ template metaprogramming allow us to create processing blocks that are very fleixible and reusable. In this case, we can at compile time specify the arguments the dispatcher will accept and compute the signature of the functions that the dispatcher can call. This frees the CPU from this job at compile time, and ensures that we cannot compose blocks that are incompatible.

4.2 IEEE 802.3 implementation

Our prototype implements a simple processing block for basic IEEE 802.3 support. The processing block encapsulates the process of adding and removing the IEEE 802.3 ethernet link-layer headers, but we currently do not support logical link control(LLC). The ethernet sublayer is supported, and the upon reception in the Receive member, the source and destination addresses are pulled out, the ethertype field determined and the request hook called with the decapsulated ethertype field. In our test implementation, we connect the Request hook to a generic dispatching block that performs a table lookup in order to find the correct receiver function, for example IPv4, ARP, or IPv6. We pass the decoded ethertype field, and it is then the responsibility of the dispatcher block to select the correct hook to call.

When sending data, we provide a simple Transmit function that encapsulates the payload in a (dst,src) pair and forwards the packet to the lower layers(ie. the driver). Once the packet is encapsulated, it is forwarded to the Request hook.

Finally, we provide a few functions to configure per-device data(physical interface addresses, etc.) and a few helpers for easily getting per-device configuration data. The helper functions are mainly for use in adaptors binding together processing blocks, eg. when performing an ARP request we need a MAC address(the "primary" address) to fill in as the sender address.

4.3 IPv4 implementation

The IPv4 suite of protocols can be implemented on top of our framework, by seperating each protocol component into a seperate processing block. These blocks can then be bound together using the hook mechanisms and dispatcher blocks.

The IPv4 protocol itself is implemented entirely as a single processing block. This was done because the protocol specifies a single header, up to 60 bytes in length, that contains all standard and optional parameters needed for processing and IPv4 frame. The flow inside the IPv4 processing block can be visualized as depicted in Figure 4.1. On the reception path, packets enter via the Receive member and get either locally delivered(via the Inform hook) or routed, while packets on the transmission path enter via the Transmit member and then get delivered towards the network(using the Request hook). The Receive function is passed a packet, along with the link-layer type that received the packet(either unicast, multicast or broadcast). The link-layer source and destination addresses are not needed, and as such are not passed in. Packet processing within the block is done by simple function calls, allowing the compiler to optimize and inline freely. By looking up internal interface-specific data, the Receive function can determine if local delivery or routing is desired. In the case of local delivery, packets are passed to the reassembly logic, which will consume the packet. Once the final fragment of a packet has been received, the packet is forwarded to the upper level layers, passing the IPv4 source and destination addreses, along with IPv4 protocol number, to the upper layers via the Inform hook. Packet forwarding simply looks up the route that will used for transmission and invokes the output logic.

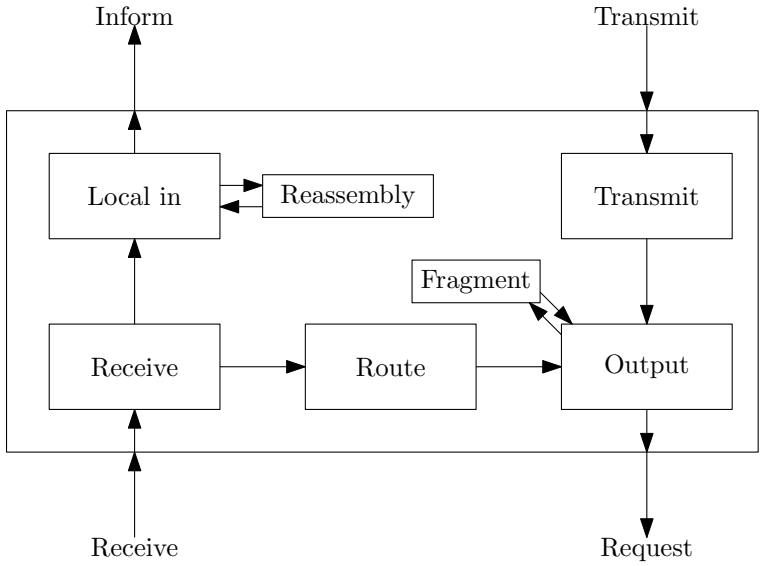

Figure 4.1: IPv4 processing block internal structure.

The Transmit function receives packets from the upper layers, as well as source and destination addresses and the protocol number of the packet that will be en-

capsulated, and encapsulates the packet. The route of the packet is determined, and he encapsulated packet is then forwarded to the output logic.

The output logic will verify that the packet does not exceed the MTU of the interface, and then forward the packet via the Request hook. If the packet does exceed the MTU, it is repeatedly split in to two, with the first fragment being shorter than the MTU, and the second being of whatever remaining size. This is repeated until the second fragment is also shorter than the MTU, at which point fragmentation terminates.

IPv4 packet reassembly is done using the `defragmenter` class. This class has two member functions, one for receiving packets and one for pruning fragmentation states. Since IPv4 reassembly introduces relatively short lived state in the receiver, we have encapsulated it in a seperate class rather than in the IPv4 packet processing block itself. The reassembly state table is implemented as a `std::map` keyed by a tuple of device identifier, source address, destination address, protocol number and the IPv4 Identification field. The map values are `gc_ptr`s to reassembly states. The `std::map` container is usually implemented as a red-black tree, allowing $O(logn)$ lookup time(but in practice cache effects may affect this), and since it is an STL container we can simply change it, to eg. a hash table(`std::unordered_map` provides one such), if the data structure proves to be a bottleneck. The map is protected by a lock that is held for a very short time while the map is modified and/or accessed and then released. Each reassembly state also has a lock that is acquired before a new fragment is attempted for reassembly in the given state. This scheme allows us better parallelism than a single lock because the global lock(around the state table) is only held for a very short time, allowing multiple threads to work on the state table, as long as they work on seperate reassembly states. The parallelism should be similar to a read-write lock at global level coupled with an exclusive lock at the state level, but the selected scheme provides better performance when new states are created. This is because a thread creating a new state does not need to wait for the completion of all threads working on single states. This would not be possible if we used nested locking, as we would need exclusive access on the top-level lock when modifying the map.

The IPv4 processing block provides some per-interface data in the form of a list of addresses associated with the interface. This data is implemented as a `gc_ptr` to a vector of addresses. This was chosen because the number of addresses assigned to a given interface will typically be low(1 address in the mast majority of cases). This means that updating the list will involve copying the vector of addresses, updating the copy and swapping the target of the `gc_ptr`. The old list will then be released after the garbage collection threshold. Since the list is updated seldomly, and the amount of data copied is small, it makes sense to optimize for the reader. This scheme allows reads to be completely lock-free when

they access the address list, provided that they only the list. One drawback of using a vector is that a vector can change size. This means that a level of indirection is used when accessing the list items(the vector must maintain a pointer to the dynamically allocated array it maintains) and a second indirection level is imposed by the `gc_ptr`. A better solution here would be to use a container that allocates its required storage at creation time, such that it can allocate a single segment of memory for data and control block. Unfortunately, C++ does not contain such a container, but the C++14 `std::dyn_array` container will allow this use case. Until then, the performance overhead of the double indirection should be low, as both the `gc_ptr`, vector control block and vector data segment will be in (sharable, per the MOESI protocol) cache in the vast majority of cases, as no writes are done to any of these locations outside of the configuration code path.

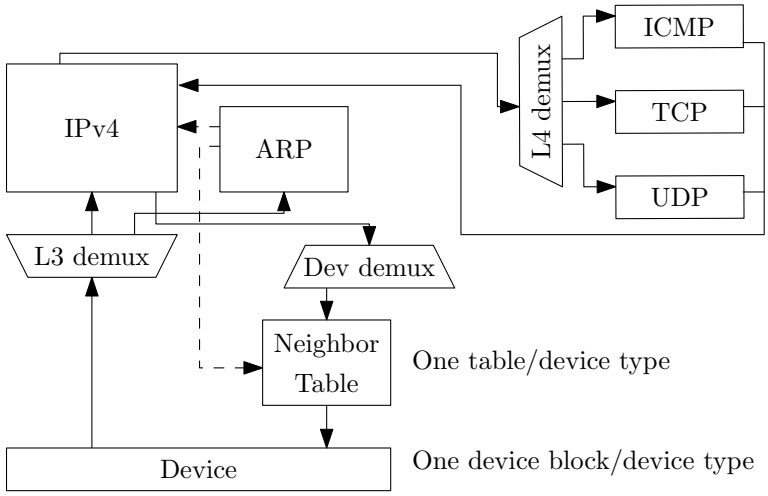

Figure 4.2: IPv4 implementation processing block overview.

In order to make IPv4 function, we need to provide interfaces for both upper and lower layer protocols. The upper layer protocol interfaces are mostly trivial. To facilitate packet encapsulation when moving down the stack, we let upper layer protocols specifiy source, destination and protocol number fields when calling the Transmit function. Transmit can then take care of performing the correct formatting and encapsulation of the packet data. When moving up the stack, our Inform hook likewise outputs the source, destination and protocol numbers. The intention is then to allow a generic dispatcher block to perform dispatch to upper layer protocols like TCP, UDP and ICMP, each providing a Receive member accepting the source and destination fields(along with packet data, of course).

Communication with lower level protocols is usually more involved. When sending IPv4 packets over IEEE 802® networks, we need to send the packet to the destination MAC address of the host that corresponds to IP address of the next hop(a directly connected host). This is done using the Address Resolution Protocol(ARP), by sending an ARP solicit message and waiting for a response. We send a broadcast ARP packet, asking for the MAC address that corresponds to a given IP address, and the host owning the given IP address replies with an ARP reply containing its MAC address. Generally, we want to buffer at least one(but preferaby a few) packets of data while we wait for the ARP reply to come back. To facilitate this, we've implemented a generic neighbor table data structure with a generic neighbor data structure for every neighbor entry. We use a two-stage locking approach, for similar reasons and with the same characteristics as the one used in the IPv4 reassembly module. The neighbor table and neighbor data structures are template classes and must be instantiated with a "Provider" class. This allows us to implement new neighbor table management schemes, allowing us to reuse the neighbor table classes for other protocols, eg. IPv6. The provider used for IP over IEEE 802® networks is aptly named arp_provider, and this provider itself is templated with the types of the upper(IPv4) and lower(IEEE 802®) address types, along with an "info" class that allows it to find the physical layer address that will be used when broadcasting(the broadcast MAC address) ARP requests. The provider class provides a method for sending ARP requests(the `solicit` function) as well as a function for sending IPv4 encapsulated frames to the network. Since the neighbor table data structures are all template classes, they are resolved at compile time. This eliminates the need for runtime polymorphism,

The other side of the ARP functionality is implemented by the ARP receiver processing block. This block provides a single member function Receive, used for receiving ARP packets. It also provides a few hook, namely:

Request For receiving ARP packets from the network.

`is_local_address` For determine if a given IP address is local to our host and interface.

`primary_address` For retrieving the primary physical address for the given interface.

`update_neighbor` For performing an actual neighborhood table update.

These hooks allow the ARP receiver block to interface directly with data neighbor table data structure, but without tightly coupling the two.

ICMP and other transport layer protocols are implemented as a seperate processing block, with a Receive function and a Request hook, allowing the Receive function to be the target of a generic dispatcher(itself the target of an IPv4 block Inform hook) and the Request hook to be bound directly to the Transmit

function of a IPv4 processing block.

A diagram showing connections between the blocks in the IPv4 implementation can be seen in Figure 4.2.

4.4 IPv6 implementation

Like the IPv4 implementation, the IPv6 protocol can be implemented as a set of processing blocks, one for each protocol in the IPv6 suite. The interfaces to the individual processing blocks, along with how they are connected, are very like the choices in IPv4, but there are some important differences.

The biggest difference occurs when handling options(called extensions in IPv6). In IPv4, options are a part of the IPv4 header and thus have to be decoded by the IPv4 processing block. In IPv6, all extensions(bar one, the hop-by-hop extension) are examined only by the end hosts of the connection. Furthermore, extensions are not actually part of the IPv6 header. Instead, the IPv6 header contains a "Next header" field that indicates the type of payload immediately following the IPv6 header. This field can contain normal protocols, such as TCP, UDP or ICMPv6, but it can also represent the next IPv6 extension header. This allows us to vastly simplify the IPv6 processing block when compared to the IPv4 block, because fragmentation, an IPv6 extension header, does not need to be handled in the IPv6 processing block, but can instead be handled just as upper layer protocols. This can be done by connecting the IPv6 Inform hook to a dispatcher, and then connecting all upper layer protocols, along with the IPv6 extension handlers, to this dispatcher. The Inform hook on each of the IPv6 extension headers can then itself be connected to the dispatcher, allowing multiple options to be processed for any one packet. The dispatcher fallback hook can be connected to a simple lambda function that sends an ICMPv6 Parameter Problem message to the sending host in case no handler for the given upper layer protocol could be found.

Another large difference is in how neighbors are handled. As in IPv4, an IPv6 enabled hosts using an IEEE 802® network needs to be able to translate logical IPv6 addresses into physical addresses. In IPv4 this was done by ARP. IPv6 has a seperate mechanism for this, known as the Neighbor Discovery Protocol(NDP). NDP is based on ICMPv6 and combines the features of ARP and and some of the features of DHCP, in order to provide neighbor address translation and auto configuration of globally reachable addresses. We first focus on the ARP-like features.

Using the same neighbor table construction as we used for IPv4 and ARP, we can easily create a neighbor table that allows translation of IPv6 addresses into MAC addresses. Since the neighbor table is a template class and needs a provider, we can simply create a NDP provider that will send Neighbor Solication NDP messages. Only a single class needs to be created, and it only needs to provide the two public methods `solicit` and `output`. This shows the power of a policy-based design for this data structure.

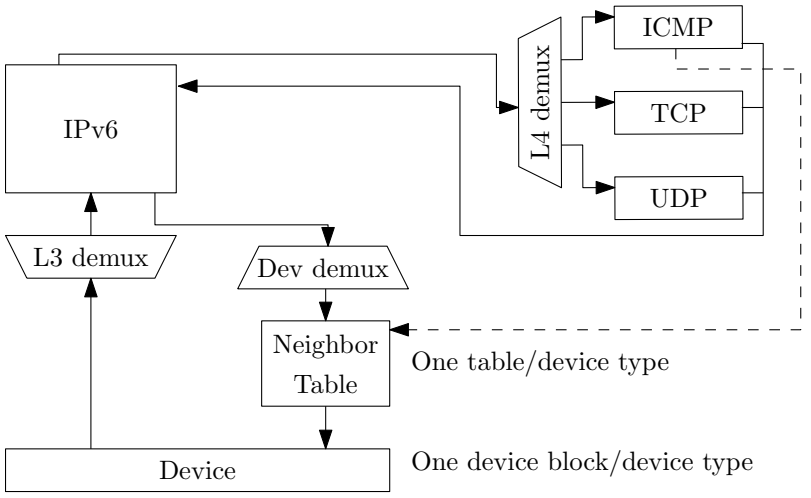

Figure 4.3: IPv6 implementation processing block overview.

Setting up IPv6 for an interface is relatively involved, and our approach would be to spawn a seperate thread to perform this task. This thread can then send and receive packets using the ICMP module, allowing a "scripted" startup of the interface. The address discovery parts of NDP could be implemented in this module in order for the interface to get a unique global address.

A diagram showing connections between the blocks in the IPv6 implementation can be seen in Figure 4.3.

4.5 Interfacing between external systems

Our network aims to be flexible, and as such can be used as an "embedded" stack inside an application, or it can be used as the central networking stack for a general purpose operating system. In the latter case, we need to determine

where to place different components of the network stack.

There are several available approaches for this, the most obvious being to emulate the behavior of modern operating systems like Linux and FreeBSD. In this scheme, we place the entire network stack in kernel-space, and using specialized processing blocks we emulate the POSIX socket interface.

It would however be interesting to reconsider the placement of various parts of the network stack. Both TCP and UDP specifiy multiplexing functionality, which means that the protocol cannot be completely handled by a single user-space process because all traffic would go to that application. To remedy this problem, we can place a demultiplexing module in the protocol stack, which only task is to generate a tuple of the type (src_ip, dst_ip, src_port, dst_port) for each packet, and based on this forward the packet to the correct user-space application. The actual TCP or UDP socket could then be implemented in the user-space application rather than the kernel.

There are multiple benefits to picking this approach over the traditional approach. The protocol stack itself is of course simplified, as a result of the smaller code required to handle only a small part of the complete protocol. A more interesting benefit is that the application developer can use a custom implementation of the UDP and TCP protocol if so desired. One could imagine a case where a specific TCP congestion control algorithm would be desired, or where specific memory handling could reduce latency. The amount of buffering present is also reduced by our approach. Normally, data is buffered between the application layer and the kernel layer, but with our approach the application could potentially busy wait on the data if extremely low latency was required. This is similar to the approach taken by several zero-copy packet reception mechanisms like netmap[32] or PF_RING[33].

The approach does have a few drawbacks, mainly in regards to security. Since the application is getting much closer to the internetworking layer than usual, one could imagine that network security could be compromised by a rogue application. This could be solved by introducing more checks inside the internetworking layers and the protocol multiplexing code. In most cases this would probably be the preferable choice regardless, as each component should in general follow the robustness principle.

POSIX sockets could be implemented as a wrapper layer on top of the user-space protocol implementation, reaping some of the benefits of having the protocol implementation in user-space. The POSIX sockets layer could also be implemented as seperate processing block(s) as upper-level targets for the protocol demultiplexer blocks. The downside of such a hybrid approach is the possible extra overhead that will be introduced on clients that use the native interface rather

than the POSIX socket layer. For this reason, along with reducing the amount of code running in kernel-space, implementing POSIX sockets as a shared library for use by the user-space applications is probably better than having it as a kernel-space implementation.

In microkernels, and systems with low overhead interprocess communication (IPC), we can take this one step further and move the network stack out of the kernel and into a seperate user-space process. This will isolate the kernel from network stack bugs and vice versa. To take this further, we can also move drivers out in seperate processes, isolating both client applications and the network stack itself from potential bugs in the driver. This requires a low-overhead IPC mechanism for performance reasons, preferably based on a zero-copy mechanism, for example implemented by moving entire pages. For concurrency reasons, it would be optimal to have multiple IPC queues between each component, that is from each driver to/from the network stack and from the network stack to/from each application. This would allow each of these to spawn a thread per CPU map each thread to a specific IPC queue. This would allow each queue to be used as a single-producer/single-consumer queue. Using a ringbuffer, this can be made extremely efficient, and would allow the system to take maximum advantage of the multi-queue network drivers.

Chapter 5

Testing

In this section we will discuss how we evaluated the Trokis networking stack prototype implementation. During development, we continually tested the design by loading a pcap-file with random data sampled from an active IPv4 network, and injecting the packets in this file into the network stack. This allowed us to quickly get feedback on implementation changes during the development cycle. Post-development, we perform a functional test and a performance test. The functional test will show if the implementation performs correctly, while the performance test will allow us to determine if the overall overhead of the Trokis approach is sound from a performance standpoint.

The code developed can be found in Appendix B.

5.1 Functional testing of implementation

We've implemented a partial test implemention of IPv4, based on the ideas presented here. Our implementation only supports IPv4(including fragmentation and reassembly), ICMP, and ARP, as the goal is to show the principles in action rather than creating a fully featured implementation.

Our test implementation also implements a very simple network card "driver", based on the Linux PF_PACKET facility, allowing direct MAC packet access. This does tie the implementation to Linux, but only this driver module is non-portable, all other parts are not tied to Linux(and were in fact developed on Mac OS X). The driver provides two functions, `send` and `poll`. These provide sending and receiving functionality, respectively. In our prototype, the Trokis main loops simply busy-waits on the receiver. When used in a real environment, the main loop would be event driven, but this would introduce further platform dependencies and we opted for the simpler code for the sake of demonstration. In our test runs, we had two Trokis mainloops running.

In order to verify the functionality of our implementation, we perform a few simple tests using ICMP echo requests(pings). Pings are a very simple way to check that all layers work correctly, as an echo request(the receive path) will require the packet to traverse the protocol stack in the "upwards" direction, passing the ethernet protocol, IPv4, and then on to the ICMP protocol. Similarly, the output path(the echo reply) will require the packet to start in the ICMP protocol, pass the IPv4 protocol, the neighborhood table lookup, ARP on at least the first packet, and finally on to the network. The test case used can be find in `functional_test/main.cpp` in Appendix B.

For all test cases, the network stack was compiled and tested using both Clang 3.3(with libc++) and g++ 4.8.1(with libstdc++), using maximum optimizations. In both cases, the results were identical and therefore we've only shown the test outputs from the Clang 3.3 compiles.

A note on the output shown in this section of the report. In cases where the textual width of the output exceeds the width of the paper, a backslash has been inserted to signify that the current line continues on the next line, with leading spaces of the next line removed. In some cases, informational lines have been removed, and this will be signified by `"-- REMOVED --"`.

In order to isolate the test from external influences, the Trokis test implementation and and testing tools are run on the same Linux machine, running Arch Linux with Linux kernel 3.3.9-1 using an Intel Core i3 M330 running at 2.13 GHz and with 4 GB of available RAM. A virtual ethernet adapter was set up using the following commands:

```
$ ip link add name side_a type veth peer name side_b
$ ip link set dev side_a up
$ ip addr add 10.0.0.9/8 dev side_a
$ ip link set dev side_b up
```

This sets up two networking devices, `side_a` and `side_b`. We will bind the prototype Trokis stack to `side_b`, and apply the IPv4 address `10.0.0.9/8` to `side_a`. The stack will listen on IP `10.0.0.10/8`.

Our first test case simply pinging our host over a network, using standard minimum-length(64 byte payload) ICMP packets as generated by the iputils[34] ping tool(as packaged by Arch Linux), with the following result:

```
% ping -c 4 -O 10.0.0.10
PING 10.0.0.10 (10.0.0.10) 56(84) bytes of data.
64 bytes from 10.0.0.10: icmp_seq=1 ttl=64 time=0.161 ms
64 bytes from 10.0.0.10: icmp_seq=2 ttl=64 time=0.034 ms
64 bytes from 10.0.0.10: icmp_seq=3 ttl=64 time=0.059 ms
64 bytes from 10.0.0.10: icmp_seq=4 ttl=64 time=0.058 ms
-- REMOVED --
```

We see that all four ICMP echo requests are answered. We also see that the first echo request is answered much slower than the three last requests. This is to be expected, as neither the Linux testing machine nor Trokis know the MAC address of the opposite side of the connection and thus an ARP transaction needs to be performed. To ensure that it is actually Trokis answering, we terminate the Trokis process, flush the test machine ARP cache and rerun the test:

```
% ping -c 4 -O 10.0.0.10
PING 10.0.0.10 (10.0.0.10) 56(84) bytes of data.
no answer yet for icmp_seq=1
no answer yet for icmp_seq=2
no answer yet for icmp_seq=3
From 10.0.0.9 icmp_seq=1 Destination Host Unreachable
From 10.0.0.9 icmp_seq=2 Destination Host Unreachable
From 10.0.0.9 icmp_seq=3 Destination Host Unreachable
From 10.0.0.9 icmp_seq=4 Destination Host Unreachable
-- REMOVED --
```

This time no replies are received, ensuring that our previous test was actually talking to Trokis.

We also wish to test fragmentation. We do this by sending a ping packet of every payload size between 56 and 5000. We do this using a small custom script `sweeper.sh`(included as Appendix A, Section A.1). We run sweeper on the Trokis prototype:

```
% ./sweeper.sh 10.0.0.10
Passed: 4945 / 4945
```

This means that Trokis has received ICMP echo request packets with paylods of all sizes from 56 to 5000 and correctly replied to them.

Finally, we perform the sweeping test again, this time under Valgrinds memcheck running full leak check mode. Memcheck is a memory error detector that allows us to check for uninitialized variables, buffer overruns, memory leaks, double frees, and more[35]. The resulting output when terminating Trokis is:

```
% sudo valgrind --leak-check=full ./trokis
-- REMOVED --
==27358== Command: ./trokis
==27358==
^CTerminating...
==27358==
==27358== HEAP SUMMARY:
==27358==     in use at exit: 56 bytes in 2 blocks
==27358==   total heap usage: 2,617,345 allocs, 2,617,343 frees, \
    2,015,421,094 bytes allocated
==27358==
==27358== 56 (8 direct, 48 indirect) bytes in 1 blocks are \
    definitely lost in loss record 2 of 2
==27358==    at 0x4C2C04B: malloc \
    (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==27358==    by 0x4085E09: operator new(unsigned long) \
    (in /usr/lib/libc++abi.so.1.0)
==27358==    by 0x40D91C: _ZNSt3__114__thread_proxyINS_\
    5tupleIJMN6trokis2gc12gc_reclaimerEFvvEPS4_EEEEEPvS9_ (thread:340)
==27358==    by 0x507ADD1: start_thread (in /usr/lib/libpthread-2.17.so)
==27358==
==27358== LEAK SUMMARY:
==27358==    definitely lost: 8 bytes in 1 blocks
==27358==    indirectly lost: 48 bytes in 1 blocks
==27358==      possibly lost: 0 bytes in 0 blocks
==27358==    still reachable: 0 bytes in 0 blocks
==27358==         suppressed: 0 bytes in 0 blocks
-- REMOVED --
```

We see that there is a 56 byte memory leak, originally allocated by the function:

```
_ZNSt3__114__thread_proxyINS_5tupleIJMN6trokis2gc12gc_\
    reclaimerEFvvEPS4_EEEEEPvS9_
```

Using c++filt, we can see what this mangled name translates to:

```
% c++filt _ZNSt3__114__thread_proxyINS_5tupleIJMN6trokis2gc12gc_\
    reclaimerEFvvEPS4_EEEEEPvS9_
void* std::__1::__thread_proxy< \
    std::__1::tuple< \
        void (trokis::gc::gc_reclaimer::*)() \
              , trokis::gc::gc_reclaimer* \
    >>(void*)
```

The ___thread_proxy function is a libc++ helper function that in this case wraps our garbage collection thread. It is unknown why the inner object allocated here is not freed, but it most likely has something to do with the exact timinig when Trokis exits, or simply a bug in libc++. To ensure that the problem is not with Trokis, but rather with low level details of the compiler/STL implementation, we recompile with g++ 4.8.1 and rerun the Valgrind test:

```
% sudo valgrind --leak-check=full ./trokis
-- REMOVED --
==32519== Command: ./trokis
==32519==
^CTerminating...
==32519==
==32519== HEAP SUMMARY:
==32519==     in use at exit: 0 bytes in 0 blocks
==32519==   total heap usage: 4,913,332 allocs, 4,913,332 frees, \
    3,825,539,326 bytes allocated
==32519==
==32519== All heap blocks were freed -- no leaks are possible
-- REMOVED --
```

This time, there are no leaks. This means that our garbage collection scheme has reclaimed all resources that were released during the execution of Trokis. We can confirm that actual garbage is being created, by disabling the garbage reclamation entirely. We do this simply by commenting out line 31 in the file `trokis/support/gc_reclaimer.hpp`:

```
30  if(ptr) {
```

```
31      // deleter(ptr); // Performs actual garbage reclamation
32  }
```

This completely disables garbage deletion, meaning that no memory is ever reclaimed by the garbage collection system. We recompile with g++ 4.8.1 and rerun the Valgrind tests. The resulting output from Valgrind(this time without full leak checking) when terminating Trokis is now:

```
% sudo valgrind ./trokis
-- REMOVED --
==19998== Command: ./trokis
==19998==
^CTerminating...
==19998==
==19998== HEAP SUMMARY:
==19998==     in use at exit: 10,059,005 bytes in 40,340 blocks
==19998==   total heap usage: 3,764,895 allocs, 3,724,555 frees, \
    2,910,237,260 bytes allocated
==19998==
==19998== LEAK SUMMARY:
==19998==    definitely lost: 3,314,480 bytes in 14,625 blocks
==19998==    indirectly lost: 6,742,677 bytes in 25,710 blocks
==19998==      possibly lost: 1,848 bytes in 5 blocks
==19998==    still reachable: 0 bytes in 0 blocks
==19998==         suppressed: 0 bytes in 0 blocks
-- REMOVED --
```

As we can see, this time the stack leaked over 10 MB of memory over the course of the run.

Based on these tests, we can confirm that garbage collection is working correctly, and that the prototype correctly replies to ICMP packets and implements ARP.

5.2 Performance testing

In order to determine if the implemented prototype could potentially support wire-speed 10 Gbps ethernet applications, we implement a simple IPv4 forwarding performance test.

In our test, we load in a number of minimum-length packets IPv4, and inject them into a network stack set up with Ethernet, IPv4 and IPv6. We then measure the amount of timed used totally for moving the packets, and calculate a packet rate. We've inject 20 million packets, corresponding to around 1.2 GB(or 1.3 seconds of minimum-size packets at 10 Gigabit Ethernet linerate, 14.88 Mpps) of traffic at the minimum ethernet packet size(64 bytes). Minimum-size packets were used in order to reduce memory overhead and reduce cache effects.

Two network devices are set up, dev1 and dev2. Each of these is assigned the IP address 10.0.x.10/24, where the "x" corresponds to the index of the device(ie. 1 or 2). A default route 0.0.0.0/0 is set up on dev2, with the nexthop address 10.0.2.1. This setup simulates a very simple packet forwarding application. The test case is implemented as a small C++ program that sets up the stack and injects data. The source code can be found in `bench/bench.cpp` in the Appendix B. The scripts `build_clang` and `build_gcc`, also found in Appendix B, can be used for building this benchmark. Note that libpcap is required, and when using clang libc++ and libc++-abi are required.

The packets are generated so that 5% are dropped at the MAC layer(they are not for our physical address), 5% are locally received at the IP layer(they for our IPv4 address), and the remaining 90% will need forwarding(they are not for our IPv4 address). Of the forwarded packets, we send 50% to a global internet address 1.2.3.4 and 50% to a local address on dev2, 10.0.2.5. Since there are no actual hosts involved in the test, we use statically configured neighbors. This data is generated using a small C++ program, `bench/generator.cpp`, found in the Appendix B. This code can be built using the `build_generator` shell script(uses clang and required libc++ and libc++-abi), also found in Appendix B.

We are not testing the multi-processor scalability of the stack, as each processing block is responsible for ensuring thread-safety. The Trokis stack infrastructure itself requires no thread-safety, it is it completely steady state. Our test thus tests the raw throughput of a simple stack with minimal, but realistic, functionality. Our test does not take into account the overhead of moving packets to and from the networking devices in the test.

We performed 100 test runs, each time measuring the amount of time taken by the packet forwarding part of the test. Thks is done by the by the shell script `bench/bench_loop`, found in Appendix B. Packet loading and memory reclamation are not part of the tested runtime. The tests were run on Linux 3.10.2-1 on an Intel Core i3 M330 running at 2.13 GHz with 4 GB of RAM. Swap was disabled to ensure the system would always keep pages in RAM. The tests were run only under clang, but g++ build scripts are also included.

We use case-resampling bootstrap to determine the true mean from our samples. A small R-script, `bootstrap_mean.r`, for calculating this can be found in Appendix A. The measured data samples are included in Appendix A. Using the R script, we calculate a bootstrap confidence interval as seen in the Table 5.1.

Table 5.1: Initial performance measurements of IPv4 forwarding test

Quantile	2.5%	mean	97.5%
Total time (ms)	7948	8058	8375
Millions of packets per second	2.52	2.48	2.39

As we can see from the measurements in Table 5.1, our network stack manages approximately 2.5 million packets per second. Assuming perfect scalability, it would require at least six CPU cores to reach the 14.88 Mpps mark, the wire-speed rate of minimal size 10 Gbps ethernet packets. Being able to optimize this would be good, and we will therefore take a look at what parts of the system are a bottleneck. For this, we will use the Linux `perf` tool.

The Linux `perf` tools provides a `record` command and a corresponding `report` command. `record` runs a program and collects samples of where the program was executing at a given time. The `report` command simply presents the samples collected by `record` these to the user. For each test, we will run the the `record` command as follows:

```
% perf record -g dwarf -f 5000 ./bench
```

We use the `-g` flag to enable stack traces and the `-F` command to specify profiling frequency of 5 kHz. After running this command, we can use the `report` subcommand to show the functions that use the most CPU resources. We run `report` as follows:

```
% perf report
```

This gives a visual overview of where the program was executing when `perf record` collected a sample. Since this is a graphical interface, we cannot efficiently show the output here, but we can describe the output and how we can solve the problems.

We see large amounts of time spent in the POSIX thread API mutex functions `pthread_mutex_lock` and `__pthread_mutex_unlock_usercnt`. Since we are

only running only a single forwarding thread, these should ideally be no-ops. We can replace them with a spinlock, allowing very low overhead in the single-threaded cases and potentially improving performance in the multi-threaded case as well.

We also see large amounts of time used in the main IPv4 receive function, `trokis::ipv4::ipv4_protocol::receive`. Using the annotation features of `perf report`, we can see a large amount of this time is spent in the IPv4 checksum calculation function, `trokis::byte_tools::checksum_ones_complement`. We can fix this by either optimizing the code(but this will most likely require us to drop to assembler) or by using hardware offloads. If we assume that the hardware can perform checksum calculations(as many network interface cards can today), we can simply skip the checksum calculations completion.

Next on the list we see the main IPv4 forwarding function. Again we use code annotation, and we can see that most time here is spent doing comparisons in the routing table. This makes sense, as our current scheme uses an extremely simple list of routes, and every route has to be checked. We can potentially reduce the amount of time spent here by sorting the routes in our list by most specific route first, but a much better solution is to simply use a better data structure like, eg. a level compressed trie.

Since improving the routing table is a larger undertaking, we will simply try to fix the first two problems: locking overhead and checksum calculation. We can fix locking overhead by using a lock type that has lower overhead. For this we implement an extremely simple spinlock and use it in place of the STL `std::mutex`. Since our spinlock implements the same interface as `std::mutex`, we can simply replace all types. No code needs to be changed.

For the checksum recalculation, we can imagine that the network interface cards will all be able to perform IP checksum offloading, and this means that we do not have to verify checksums on receive or generate correct checksums on transmit.

We implement both these changes, and the resulting code can be found in Appendix C. This code is identical to the code in Appendix B in every way but the two optimizations. We then perform the complete 100-run benchmark again. The new results are seen in Table 5.2, and the raw measurements can be found in Appendix A. As can be seen in Table 5.2, these basic optimizations resulted in considerable speedups, on an average packet rate of 44.4% over the un-optimized initial case. Further optimizations could of course be applied, but we have chosen to stop here. We will discuss this further in Chapter 7.

Table 5.2: Optimized performance measurements of IPv4 forwarding test

Quantile	2.5%	mean	97.5%
Total time (ms)	5558	5581	5644
Millions of packets per second	3.60	3.58	3.54
Speedup over initial test	42.8%	44.4%	48.1%

Chapter 6

Relation to existing solutions

In this section, we look at how different state of the art network stacks are implemented, and discuss the pros and cons of the approaches taken, and compare these solutions with our own.

6.1 Linux and FreeBSD network stacks

The Linux and FreeBSD network stacks are interesting in the context of our project because they represent the status quo of network stack design in two very popular(at least in the server market) operating systems.

A large part of the Linux and FreeBSD network stacks consists of code for managing packet data. In Linux, the structure for handling data is known as the sockets buffers, sk_buff, or just "skb" for short. The equivalent FreeBSD structure is the mbuf, short for memory buffer. The two structures are used for referencing the data that belongs to a packet, but they take very different approaches to handing the data.

The linux sk_buff[25], simply skb from here on out, consists of a series of pointers managing a buffer of packet data, along with meta data about the packet.

During processing, the meta data is filled out with pointers to the network layer header, the transport header, etc. Higher level layers use these pointers to process the packet. Parsing packets is done using packed C structs, allowing good performance but also reducing code portability. Skbs may be fragmented, in which case they point to several blocks of data. Skbs also include next and previous pointers, allowing them to be efficiently used in intrusive double linked lists. This is heavily used in the Linux kernel, for example when queueing packets.

The mbufs used in FreeBSD take a different approach[24]. Like skbs, mbufs handle potentially fragmented data, but unlike skbs they do not store large amount of metadata about the packet they are transporting. Instead, the FreeBSD stack passes the required meta data as function call arguments between layers of the network stack.

The Trokis `packet_data` structure is used for the same purpose as mbufs and skbs, supporting both fragmented and non-fragmented payloads in a common API. The goal of all these structures is to provide high performance handling of packet data, preferably with as little copying as possible. When handling actual data, FreeBSD and Linux generally use C structs and perform a cast from the underlying data buffer to the struct. This has implications for portability, as the layout and padding of C structs is implementation defined. Trokis instead picks a more portable but presumably slightly lower performance choice, of using the `packet_data` get and set helpers. This also frees the programmer from having to verify that enough contiguous data can be read.

The Linux and FreeBSD network stacks are both written in pure C, and the code is generally very advanced because it has to run in kernel mode, with some parts potentially running in interrupt context. The code is structured using functions that pass data up and down the stack, moving metadata using either skbs(Linux) or function call arguments(FreeBSD). In Trokis we use a similar approach to FreeBSD, passing almost all data using function call arguments. Unlike both Linux and FreeBSD, the processing blocks of our stack do not have a real concept of "up" and "down" the stack, as this is implicit in the semantics of the function call(ie. Receive sends data up, Transmit sends data down). Another difference is that Trokis processing blocks perform all calls to external functions through hooks, rather than as direct functions.

For dispatching protocols, both Linux and FreeBSD implement lookup tables for each dispatch layer. In Trokis, this is generally handled using the generic dispatching block, but a custom block can of course be used if required.

Linux and FreeBSD both implement the POSIX sockets API on top of their networking stacks. As discussed in Chapter 4, this could be implemented on

top of Trokis as a (set of) processing blocks.

6.2 The Click Modular Router

The Click Modular Router[2] takes a radically different approach when compared to modern operating system network stacks. In Click, the entire router configuration is modeled as a directed graph of processing blocks known as "elements". Each element possesses a set of input and output ports, and each output port can be matched up with a corresponding input port on another element. Outputs know what element they are connected to, and inputs know what outputs connect to them. This allows packets to be either pushed from an element(pushed out of an output) or pulled to an element(pulled in from an input). Using this scheme, Click allows extreme degrees of flexibility, and almost any packet processing configuration can be realized. A Click configuration can be loaded on-the-fly to replace a previously running configuration, allowing high runtime flexibility in the router configuration.

The Click Modular Router infrastructure is programmed in object-oriented C++. Each element is represented by a seperate object, deriving from a common Element class. The Element class provides the infrastructure for the push and pull based processing, while the derived class implements the specific element capability, for example IPv4 routing, ARP processing or IP packet Time-to-live decrement. In this standard setup, moving packets between elements requires either one or two virtual calls[2]. Virtual calls prevent the compiler from optimizing across them, and also has performance implications from a CPU point of view. This coupled with the fact that Click elements are typically very generic in order to maximize flexibility, introduces a potential performance bottleneck. To alleviate this, Click includes a tool, click-devirtualize, that can "compile" Click configuration files to C++ source files, creating a new C++ class for every element in the configuration. The created C++ classes have specialized port interfaces that directly point to the correct element type connected to the port, allowing the compiler maximum freedom in optimizing calls between elements.

The data pushed between elements is encapsulated by a Packet object. The same packet contents can be pointed to by several Packet objects, and modifying the actual data is allowed iff a single packet references the data. This approach minimizes packet copies and is overall a very simple and elegant system for managing packet data.

The Click architecture also specifies an elaborate scheduling system, allowing elements to be scheduled either implicitly(through push or pull processing) or

explicitly(through an element scheduler). This allows all sorts of element types to be created, including token buffers(rate-limiting queues), queueing disciplines, polling elements, etc.

The Click infrastructure can run in user-space or as a kernel model, with the intent of the latter being to achieve better performance than can be achieved in user space(although recent developments, eg. netmap[32], allow user-space performance at least on par with that of kernel-mode code).

Click only supports polling mode, meaning that each network interface in the configuration is checked for new packets at regular intervals. Interrupts are not supported. This ensures that a Click configuration will never livelock, but can introduce latency(especially under light workloads).

Click is especially useful in testing deployments, for example when simulating large networking systems.[1] The high flexibility allows Click to be used in very diverse applications, ranging from "simple" packet routing to advanced network simulation with latency and packet loss simulations. Click is also the basis of the RouteBricks[36] project, where it is used to power the packet processing of clustered Intel® Xeon® servers, at speeds of in excess of 40 Gbps.

Comparing to Trokis, the "elements" in Click are in almost direct correspondance with the "processing blocks" of Trokis, but there are some important differences, mainly being that Trokis has no (built in) concept of block scheduling or "ports". Instead, Trokis allows arbitrary "ports"(hooks and members) to be defined and typechecked by the compiler, and scheduling is implemented outside of Trokis if required. This allows the programmer to pick much more freely what features they need. Click ports also support only a single parameter, namely the packet that is to be moved. This means that any data passed between blocks has to be passed as "annotations", extra data associated with a given packet in Click. In our approach, we instead rely on passing arguments between functional blocks.

The Click data structure for passing data, Packet, is similar to a combination of the Trokis `packet` and `packet_data` structures, but has some extra features. The Click Packet structure supports copy-on-write, allowing a block to send a copy of a packet to multiple output ports. In Trokis, packets are always exclusive to a single owner, and are handed between processing blocks using C++ move semantics, forcing the caller to give up ownership of the packet. There is however nothing preventing a copy-on-write extension to the Trokis `packet_data` structure. This could potentially be done without breaking any existing code,

[1] At Open Source Days 2013, Lars Bro from Siemens talked about how Siemens used Click to simulate network traffic for a large-scale networking solution for the Danish S-Tog train lines in the Copenhagen area. The Click configuration used had several hundred thousand processing elements.

because of the high level of abstraction of the `packet_data` structure.

Overall, Trokis is meant to perform only part of what Click does, since Click provides a complete infrastructure while Trokis focuses on actual packet processing.

6.3 Lightweight ANA

The Lightweight Autonomic Network Architecture[3], LANA, is, like Click, an extremely flexible networking stack, but it takes a radically different approach to modularity. The LANA implementation runs in the Linux kernel and interfaces with the Linux network layer, allowing it to exploit the drivers already present in the kernel.

The LANA project is really a lightweight architecture inspired by the Autonomic Network Architecture(ANA), thus the name Lightweight ANA. The two share many parts of their philosophy and architecture, but we have chosen to focus on LANA here, as the ANA project has goals that differ greatly from those of our project.

Like Click, LANA is based on the idea of processing elements, in this case called Functional Blocks(FBs). Each FB consists of two handlers, one for packets and one for events(out-of-band information about state changes in the system), along with various metadata, eg. a name and block-local data. FBs are addresses only by an opaque identifier known as an Information Dispatch Point(IDP). The list of all IDPs is managed centrally by an entity known as the Functional Block Registry, allowing other blocks(and outside systems) to look up the IDP to use when they wish to deliver packets to a FB.

Trokis processing blocks, unlike LANA FBs, allow arbitrary inputs and outputs rather than just the two allowed by LANA. This allows a cleaner programming model when composing functional blocks, but strictly speaking the two approaches should be equally powerful. Trokis does not use IDP, or a similar mechanism, but instead uses hooks for its indirections. This allows most of the flexibility of the LANA IDP approach, but a LANA block can push multiple IDPs on to the IDP stack, allowing a block to inject a pipeline of targets to be called in succession, rather than just a single target(as is possible with Trokis). It is unknown whether or not this functionality is useful in practice.

Packet delivery is driven by a loop that continously fetches and forwards pending packets to FBs. A FB packet handler can specify the next IDP that will receive

the packet by manipulating a list of IDPs associated with the packet. Once returning the packet to the loop, the loop pulls the next IDP from the packet IDP list and forwards the packet again, until the packet is consumed by a FB, at which point the packet loop simply continues to the next packet. The IDP to FB lookup is driven by an intrusive crit-bit tree, for performance reasons.

Trokis does not enforce any particular delivery mechanism as such, but once a packet has been delivered to a processing block has been delivery to a block, the processing block may decide to to drop, buffer or forward the packet as it deems required. Trokis also avoids the IDP to FB indirection(an $O(m)$ operation, where m is the number of bits in an IDP)[3], by instead using a function pointer(`std::function`). Since the IDP to FB indirection itself produces a function pointer(to the function block packet handler), storing a function pointer will always be faster. LANA does however allow new instances of functional blocks to be created at runtime, something that Trokis does not support.

The LANA architecture scales very well[3] and allows great flexibility, but unlike Click it only supports push-style processig of packets. This is not necessarily a bad thing, but it does mean that some elements, like token buffers and queueing disciplines, potentially would have to use outside functionality, for example using the beforementioned event hooks to signal rate-limiting to the previous elements in a processing chain. An example of a place where this could be required is when outputting data onto a loaded network interface. If the network interface card is fully utilized, attempting to send extra traffic to the device buffers will fail as there is simply no room. In such cases, the protocol stack may want to buffer packets without dropping them. The LANA architecture does not allow this per-se, as packets are returned to the packet delivery loop, and a given FB therefore has no way to known the return value of the next FB in the chain. This could potentially be resolved using some sort of event, where the network interface sends an event to the sending FB in cases where the interface cannot queue more data.

Trokis also only provides push-style processing, and implementing token buffers or queueing therefore has the same implications in Trokis as in LANA. Trokis functions and hooks should however make it much easier to implement the required control loop between processsing blocks, whereas LANA blocks must multiplex this functionality on top of the event handler.

Memory handling for packets in LANA is done using the normal sk_buff infrastructure in the Linux kernel, see Section 6.1 for details.

Chapter 7

Conclusion

In this project, we have shown that the use of modern features in a language like C++11, a network stack can be implemented as a set of highly decoupled modular processing blocks. The use of member functions and hooks allows components to be swapped out without changing the internals of the connected processing blocks. Even blocks with seemingly incompatible member/hook function call signatures can be connected easily with the help of C++11 lambdas, allowing a more natural composition than the virtual function call approach.

We've shown that it is possible to create a minimal set of infrastructure components to allow easy implementation of the processing blocks in a system. We've implemented basic packet payload handling, device identifiers, a basic garbage collection scheme and thread-safe, high-performance, safe memory reclamation.

Currently, composing functional blocks is relatively verbose, requiring `std::bind` expressions or spelling out the entire types in the lambda functions that are used as hooks. This is because of the C++11 language, but with C++14 some of these problems will be resolved with the help of polymorphic lambda functions. A solution that could be implemented now would be to extend the processing block infrastructure to not allow direct access to member functions, but instead expose functions for returning functors that could be registered directly to the `std::function` hooks.

The performance of our approach is currently lower than we would like. Ideally, we would like achieve at the ethernet line-rate with at most four cores on our test machine. Our test machine was a relatively low-powered one, an Intel i3 2.13 GHz laptop, but the overhead of performing the simple packet processing in our test should not be that large.

Even after optimizing, our test network stack could only achieve 3.58 Mpps per core on the test machine. We believe that most of the time is currently spent in packet data helpers. This is based on the fact that these are currently using naive, unoptimized code. For example, when reading a 32-bit(4-byte) integer the current code reads four individual bytes and sums these together using additions and bit shifts. On a CPU that can support 32-bit integers natively, it will be many times faster to simply read the 32-bit integer in a single instruction. The upside of the approach taken is that it works no matter how the memory of the lower level container is laid out, but in our case we are using simple `std::vector`s that are guaranteed to be contiguous in memory. The extra genericity is thus resulting in a drop in performance.

Overall however, we feel that we have succeeded in implementing the basics of a flexible network stack that can be used in applications ranging from small to large. Building the functional test application using clang results in a 255kB executable, most of which is the C++ library, and this could most likely be reduced even further by using size optimizations and avoiding inlining. When running(on the test system), the functional test application starts out using 1388 bytes of resident memory, and after performing the ping sweep test the application uses 2168 bytes of resident memory(note that the entries in the fragment buffer are currently not removed). This makes the stack applicable to low-resource embedded systems. The performance is not as good as expected, presumably because of the packet decoding helpers, but even on our low-end test machine we achieved 3.58 Mpps on a single core, and on a high-end machine we could conceivably reach in excess of 4 Mpps per core(the test machine has a 2.13 GHz CPU, but 3 GHz CPUs are readily available today).

Chapter 8

Future work

While Trokis in its current form implements the basic framework for a network stack, along with a simple IPv4 implementation, there are still many different areas that can be improved. Here is a list of some potential areas, in no particular order:

- The type-safe packet decoding facility needs to be optimized so that it better utilizes the CPU, for example by using CPU instructions for swapping byte orders and using a single read for loading integers of sizes up to the CPU supported bit-width.
- Implement fragmented packet handling, so that the overhead of appending or prepending data(and headers) to packets can be minimizes. This is especially important when handling large packets, in order to avoid copies.
- Implement support for hardware offload features. Our performance tests showed that lots of time was spent performing checksum calculations, and this could be solved with the help of hardware offloads.
- Implement a full implementation of IPv6. Our current stack merely implements the header encapsulation, but no other parts are currently supported.
- Implement more IPv4 protocols, along with the demultiplexers discussed in Section 4.5.
- Implement full IPv4 options support.

Appendix A

Test code and data

Included here is source code for various test cases, along with resulting data.

A.1 sweeper.sh

This is the shell script used to send ICMP echo requests at increasing payload size.

```
#!/bin/sh

let "pass = 0"
let "count = 0"
IP=$1

for size in {56..5000}
do
    ping -c 1 -s $size $IP &> /dev/null
    RETVAL=$?
    if [ "$RETVAL" != "0" ]; then
        echo "Fail!"
```

```
    else
        let "pass = pass + 1"
    fi
    let "count = count + 1"
done
echo "Passed: $pass / $count"
```

A.2 bootstrap_mean.r

This is the R script used to calculate the mean and confidence intervals of the performance benchmark.

```
# Constants
rounds = 100000 # Number of bootstrap rounds(Monte-Carlo case resampling)

# Load numbers from results file. Each result should have a seperate line.
data <- scan('results.txt')

# Perform bootstrap resampling
means <- sapply(c(1:rounds), function(x) \
    sample(data, length(data), replace=TRUE))

# Output mean and standard deviation
cat(sprintf("Mean: %f", mean(means)))
cat(sprintf("SD:   %f", sd(means)))

# Find confidence interval on the bootstrapped data
confi <- quantile(means, probs = c(0.025, 0.975))
cat(sprintf("95%% conf. interval: %f <= %f <= %f", \
    confi[1], mean(means), confi[2]))
```

A.3 Performance benchmark initial data set

This is the data from the initial performance benchmark. All results are in milliseconds per 20 million packets.

```
8220 8097 8094 8039 7960 8056 8195 8026 7974 8090
```

```
7954 8034 7950 7948 8019 7949 7953 7943 8050 8375
8199 7955 7951 8007 7948 8138 8034 8082 8049 7949
8083 7956 8198 7961 8130 8148 7969 8095 7948 8135
7947 8068 8014 7967 8039 8095 8079 7952 8030 7976
8144 7949 7954 8059 8110 8124 8034 8082 8091 8027
7958 8104 8043 7966 8088 8112 7975 8063 7968 8144
8090 7992 8099 8069 8139 8006 8226 8121 8434 8111
8030 7960 8239 8032 8077 8092 8053 8015 8049 7958
8250 8059 8085 8080 7965 8595 7988 8012 7961 7992
```

A.4 Performance benchmark optimized data set

This is the data from the performance benchmark on the optimized network stack. All results are in milliseconds per 20 million packets.

```
5540 5735 5564 5576 5572 5577 5574 5599 5575 5572
5640 5572 5569 5558 5588 5558 5564 5596 5566 5589
5573 5600 5572 5596 5566 5574 5585 5575 5568 5574
5610 5573 5583 5572 5579 5571 5563 5564 5578 5577
5571 5573 5574 5568 5571 5580 5574 5582 5585 5566
5573 5565 5576 5569 5564 5625 5578 5569 5562 5573
5572 5585 5566 5644 5574 5595 5626 5570 5602 5609
5566 5567 5593 5609 5575 5566 5565 5575 5568 5576
5772 5567 5572 5568 5573 5564 5604 5572 5571 5568
5577 5572 5604 5581 5575 5583 5563 5573 5567 5565
```

Appendix B

Trokis test implementation

Listing B.1: bench/bench.cpp

```
#include <iostream>
#include <cstdint>
#include <vector>
#include <array>
#include <algorithm>
#include <functional>
#include <chrono>
#include <unordered_map>
#include <map>
#include <tuple>
#include <atomic>
#include <fstream>

#include <iomanip>
#include <sstream>
#include <type_traits>
#include <iterator>

template<std::size_t N>
std::ostream& operator<<(std::ostream& os, const std::array<uint8_t, N>& array) {
    bool first = true;
    for(auto p : array) {
        if(!first) {
            os << ".";
        }
```

```
26          os << (int)p;
27          first = false;
28      }
29      return os;
30  }
31
32
33  #include "trokis/types.hpp"
34  #include "trokis/packet_data.hpp"
35  #include "trokis/support/byte_tools.hpp"
36  #include "trokis/packet.hpp"
37  #include "trokis/support/dispatcher.hpp"
38  #include "trokis/protocols/ipv4/ipv4.hpp"
39  #include "trokis/protocols/ieee802/ieee802_3.hpp"
40  #include "trokis/protocols/ipv4/arp.hpp"
41
42  #include "trokis/support/pcap_output.hpp"
43
44  #include <pcap.h>
45
46  using namespace trokis;
47
48  std::atomic<bool> running { true };
49
50  bool load_packets(std::string filename, std::vector<packet_ptr>&
        packets) {
51      char errbuf[PCAP_ERRBUF_SIZE];
52      auto pcap = pcap_open_offline(filename.c_str(), &errbuf[0]);
53      if(pcap == nullptr) {
54          std::cerr << "Failed to open pcap dump: " << errbuf << std
                ::endl;
55          return false;
56      }
57
58      while(1) {
59          struct pcap_pkthdr hdr;
60          auto pdata = pcap_next(pcap, &hdr);
61          if(pdata == nullptr) break;
62          auto pkt = std::make_unique<packet>();
63          auto pkt_data = std::make_unique<packet_data>(pdata, pdata
                + hdr.caplen);
64          pkt->replace_data(std::move(pkt_data));
65          packets.push_back(std::move(pkt));
66      }
67      return true;
68  }
69
70  __attribute__((noinline)) void run_benchmark(std::vector<packet_ptr
        >& packets, ieee802::ethernet_802_3& layer_mac, device_id devid
        ) {
71      std::size_t totcount = 0;
72      std::size_t pktcount = 0;
73      auto start = std::chrono::high_resolution_clock::now();
74      for(auto& pkt : packets) {
75          pkt->dev = devid;
```

```
76          pkt->worker = 1;
77          if(layer_mac.receive(std::move(pkt)) == packet_verdict::
                success) {
78              ++pktcount;
79          }
80          ++totcount;
81      }
82      auto end = std::chrono::high_resolution_clock::now();
83      auto diff = std::chrono::duration_cast<std::chrono::
            milliseconds>(end - start);
84      std::cerr << "Injected " << pktcount << "/" << totcount << "
            packets in " << diff.count() << " ms" << std::endl;
85      std::cout << diff.count() << std::endl;
86  }
87
88  int main(int argc, char *argv[]) {
89      /*** Initialize network stack ***/
90      using namespace std::placeholders;
91
92      /* Set up MAC layer for ethernet */
93      ieee802::ethernet_802_3 layer_mac;
94
95      /* Set up receive path ethertype dispatch */
96      utilities::dispatcher<packet_verdict(packet_ptr&&, ieee802::
            ethertype, address_type)> dispatch;
97      dispatch.set_fallback([](packet_ptr&&, ieee802::ethertype,
            address_type) { return packet_verdict::dropped; });
98      layer_mac.set_hook_inform(std::ref(dispatch));
99      layer_mac.set_hook_request([](packet_ptr&&) { return
            packet_verdict::success; });
100
101     ipv4::ipv4_protocol layer_ipv4;
102     layer_ipv4.set_gateway(true);
103
104     utilities::dispatcher<packet_verdict(packet_ptr&&, ipv4::
            protonum_t, ipv4::address_t, ipv4::address_t)>
            ipv4_dispatch;
105     ipv4_dispatch.set_fallback([](packet_ptr&&, ipv4::protonum_t,
            ipv4::address_t, ipv4::address_t) {
106         return packet_verdict::dropped;
107     });
108
109     layer_ipv4.set_hook_inform(std::ref(ipv4_dispatch));
110     neighbor::neighbor_table<neighbor::arp_provider<ipv4::address_t
            ,ieee802::dev_info_802_3>> neigh_ipv4_ethernet;
111     neigh_ipv4_ethernet.get_provider().set_output_hook(std::bind(&
            decltype(layer_mac)::transmit, &layer_mac, _1, _2, _3));
112     neigh_ipv4_ethernet.get_provider().set_primary_address_hook
            ([&](device_id devid) {
113         return std::make_pair(layer_ipv4.get_primary_address(devid)
                , layer_mac.get_primary_address(devid));
114     });
115     // Remember to switch to a demultiplexer here, if we start
            using lower level protocols that don't use MAC addresses.
```

```
116     layer_ipv4.set_hook_request(std::bind(&decltype(
            neigh_ipv4_ethernet)::transmit, &neigh_ipv4_ethernet, _1,
            _2));
117
118     /* Set up ARP receiver */
119     arp::arp_receiver layer_arp;
120     layer_arp.set_hook_request([&] (packet_ptr&& pkt, ieee802::
            mac_address src, ieee802::mac_address dst) {
121         pkt->data().push(2);
122         pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
123         return layer_mac.transmit(std::move(pkt), src, dst);
124     });
125     layer_arp.set_primary_address_hook(std::bind(&decltype(
            layer_mac)::get_primary_address, &layer_mac, _1));
126     layer_arp.set_is_local_address_hook(std::bind(&decltype(
            layer_ipv4)::is_local_address, &layer_ipv4, _1, _2));
127     layer_arp.set_update_neighbor_hook([&](device_id devid, ipv4::
            address_t upper, ieee802::mac_address lower, bool
            may_create) {
128         decltype(neigh_ipv4_ethernet.get(devid, upper)) neighbor;
129         if(may_create) {
130             neighbor = neigh_ipv4_ethernet.get_or_create(devid,
                    upper);
131         } else {
132             neighbor = neigh_ipv4_ethernet.get(devid, upper);
133         }
134         if(neighbor == nullptr) return false;
135         neighbor->set_lower_address(lower, ieee802::mac_address());
136         return true;
137     });
138
139     /* Set up ICMPv4 */
140     ipv4::icmp4::icmp4_protocol layer_ipv4_icmp;
141     layer_ipv4_icmp.set_hook_request(std::bind(&decltype(layer_ipv4
            )::transmit, &layer_ipv4, _1, _2, _3, _4));
142     ipv4_dispatch.set_handler(1, std::bind(&decltype(
            layer_ipv4_icmp)::receive, &layer_ipv4_icmp, _1, _2, _3));
143
144     // Set up dispatcher
145     dispatch.set_handler(0x0800, std::bind(&ipv4::ipv4_protocol::
            receive, &layer_ipv4, _1, _2));
146     dispatch.set_handler(0x0806, std::bind(&arp::arp_receiver::
            receive, &layer_arp, _1, _2));
147
148     // Set up devices
149     device_id dev1 = 1;
150     device_id dev2 = 2;
151     {
152         auto dev = layer_mac.get_or_create_dev_data(dev1);
153         dev->unicast_address_add({{0x01,0x23,0x45,0x67,0x89,0xab}})
                ;
154
155         auto v4 = layer_ipv4.get_or_create_dev_data(dev1);
156         v4->local_address_add({{10, 0, 1, 10}});
157
```

```
158     }
159     {
160         auto dev = layer_mac.get_or_create_dev_data(dev2);
161         dev->unicast_address_add({{0x00,0x23,0x14,0x30,0xce,0x74}})
                ;
162 
163         auto v4 = layer_ipv4.get_or_create_dev_data(dev2);
164         v4->local_address_add({{10, 0, 2, 10}});
165 
166         auto neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,5}}));
167         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
168         neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,1}}));
169         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
170     }
171     layer_ipv4.get_routing_table().add_route({{{10,0,1,0}}, 24,
            true, {{0,0,0,0}}, dev1});
172     layer_ipv4.get_routing_table().add_route({{{10,0,2,0}}, 24,
            true, {{0,0,0,0}}, dev2});
173     layer_ipv4.get_routing_table().add_route({{{0,0,0,0}}, 0, false
            , {{10,0,2,1}}, dev2});
174 
175     // Load test data
176     std::vector<packet_ptr> packets;
177     if(!load_packets("test.pcap", packets)) {
178         return 1;
179     }
180     std::cerr << "Loaded " << packets.size() << " packets" << std::
            endl;
181 
182     run_benchmark(packets, layer_mac, dev1);
183 
184     return 0;
185 }
```

Listing B.2: bench/bench.cpp

```
1  #include <iostream>
2  #include <cstdint>
3  #include <vector>
4  #include <array>
5  #include <algorithm>
6  #include <functional>
7  #include <chrono>
8  #include <unordered_map>
9  #include <map>
10 #include <tuple>
11 #include <atomic>
12 #include <fstream>
13 
14 #include <iomanip>
15 #include <sstream>
```

```
16 #include <type_traits>
17 #include <iterator>
18
19 template<std::size_t N>
20 std::ostream& operator<<(std::ostream& os, const std::array<uint8_t
       , N>& array) {
21     bool first = true;
22     for(auto p : array) {
23         if(!first) {
24             os << ".";
25         }
26         os << (int)p;
27         first = false;
28     }
29     return os;
30 }
31
32
33 #include "trokis/types.hpp"
34 #include "trokis/packet_data.hpp"
35 #include "trokis/support/byte_tools.hpp"
36 #include "trokis/packet.hpp"
37 #include "trokis/support/dispatcher.hpp"
38 #include "trokis/protocols/ipv4/ipv4.hpp"
39 #include "trokis/protocols/ieee802/ieee802_3.hpp"
40 #include "trokis/protocols/ipv4/arp.hpp"
41
42 #include "trokis/support/pcap_output.hpp"
43
44 #include <pcap.h>
45
46 using namespace trokis;
47
48 std::atomic<bool> running { true };
49
50 bool load_packets(std::string filename, std::vector<packet_ptr>&
       packets) {
51     char errbuf[PCAP_ERRBUF_SIZE];
52     auto pcap = pcap_open_offline(filename.c_str(), &errbuf[0]);
53     if(pcap == nullptr) {
54         std::cerr << "Failed␣to␣open␣pcap␣dump:␣" << errbuf << std
               ::endl;
55         return false;
56     }
57
58     while(1) {
59         struct pcap_pkthdr hdr;
60         auto pdata = pcap_next(pcap, &hdr);
61         if(pdata == nullptr) break;
62         auto pkt = std::make_unique<packet>();
63         auto pkt_data = std::make_unique<packet_data>(pdata, pdata
               + hdr.caplen);
64         pkt->replace_data(std::move(pkt_data));
65         packets.push_back(std::move(pkt));
66     }
```

```
67      return true;
68  }
69
70  __attribute__((noinline)) void run_benchmark(std::vector<packet_ptr
        >& packets, ieee802::ethernet_802_3& layer_mac, device_id devid
        ) {
71      std::size_t totcount = 0;
72      std::size_t pktcount = 0;
73      auto start = std::chrono::high_resolution_clock::now();
74      for(auto& pkt : packets) {
75          pkt->dev = devid;
76          pkt->worker = 1;
77          if(layer_mac.receive(std::move(pkt)) == packet_verdict::
                success) {
78              ++pktcount;
79          }
80          ++totcount;
81      }
82      auto end = std::chrono::high_resolution_clock::now();
83      auto diff = std::chrono::duration_cast<std::chrono::
            milliseconds>(end - start);
84      std::cerr << "Injected " << pktcount << "/" << totcount << "
            packets in " << diff.count() << " ms" << std::endl;
85      std::cout << diff.count() << std::endl;
86  }
87
88  int main(int argc, char *argv[]) {
89      /*** Initialize network stack ***/
90      using namespace std::placeholders;
91
92      /* Set up MAC layer for ethernet */
93      ieee802::ethernet_802_3 layer_mac;
94
95      /* Set up receive path ethertype dispatch */
96      utilities::dispatcher<packet_verdict(packet_ptr&&, ieee802::
            ethertype, address_type)> dispatch;
97      dispatch.set_fallback([](packet_ptr&&, ieee802::ethertype,
            address_type) { return packet_verdict::dropped; });
98      layer_mac.set_hook_inform(std::ref(dispatch));
99      layer_mac.set_hook_request([](packet_ptr&&) { return
            packet_verdict::success; });
100
101     ipv4::ipv4_protocol layer_ipv4;
102     layer_ipv4.set_gateway(true);
103
104     utilities::dispatcher<packet_verdict(packet_ptr&&, ipv4::
            protonum_t, ipv4::address_t, ipv4::address_t)>
            ipv4_dispatch;
105     ipv4_dispatch.set_fallback([](packet_ptr&&, ipv4::protonum_t,
            ipv4::address_t, ipv4::address_t) {
106         return packet_verdict::dropped;
107     });
108
109     layer_ipv4.set_hook_inform(std::ref(ipv4_dispatch));
```

```
110     neighbor::neighbor_table<neighbor::arp_provider<ipv4::address_t
            ,ieee802::dev_info_802_3>> neigh_ipv4_ethernet;
111     neigh_ipv4_ethernet.get_provider().set_output_hook(std::bind(&
            decltype(layer_mac)::transmit, &layer_mac, _1, _2, _3));
112     neigh_ipv4_ethernet.get_provider().set_primary_address_hook
            ([&](device_id devid) {
113         return std::make_pair(layer_ipv4.get_primary_address(devid)
                , layer_mac.get_primary_address(devid));
114     });
115     // Remember to switch to a demultiplexer here, if we start
            using lower level protocols that don't use MAC addresses.
116     layer_ipv4.set_hook_request(std::bind(&decltype(
            neigh_ipv4_ethernet)::transmit, &neigh_ipv4_ethernet, _1,
            _2));
117
118     /* Set up ARP receiver */
119     arp::arp_receiver layer_arp;
120     layer_arp.set_hook_request([&] (packet_ptr&& pkt, ieee802::
            mac_address src, ieee802::mac_address dst) {
121         pkt->data().push(2);
122         pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
123         return layer_mac.transmit(std::move(pkt), src, dst);
124     });
125     layer_arp.set_primary_address_hook(std::bind(&decltype(
            layer_mac)::get_primary_address, &layer_mac, _1));
126     layer_arp.set_is_local_address_hook(std::bind(&decltype(
            layer_ipv4)::is_local_address, &layer_ipv4, _1, _2));
127     layer_arp.set_update_neighbor_hook([&](device_id devid, ipv4::
            address_t upper, ieee802::mac_address lower, bool
            may_create) {
128         decltype(neigh_ipv4_ethernet.get(devid, upper)) neighbor;
129         if(may_create) {
130             neighbor = neigh_ipv4_ethernet.get_or_create(devid,
                    upper);
131         } else {
132             neighbor = neigh_ipv4_ethernet.get(devid, upper);
133         }
134         if(neighbor == nullptr) return false;
135         neighbor->set_lower_address(lower, ieee802::mac_address());
136         return true;
137     });
138
139     /* Set up ICMPv4 */
140     ipv4::icmp4::icmp4_protocol layer_ipv4_icmp;
141     layer_ipv4_icmp.set_hook_request(std::bind(&decltype(layer_ipv4
            )::transmit, &layer_ipv4, _1, _2, _3, _4));
142     ipv4_dispatch.set_handler(1, std::bind(&decltype(
            layer_ipv4_icmp)::receive, &layer_ipv4_icmp, _1, _2, _3));
143
144     // Set up dispatcher
145     dispatch.set_handler(0x0800, std::bind(&ipv4::ipv4_protocol::
            receive, &layer_ipv4, _1, _2));
146     dispatch.set_handler(0x0806, std::bind(&arp::arp_receiver::
            receive, &layer_arp, _1, _2));
147
```

```
148     // Set up devices
149     device_id dev1 = 1;
150     device_id dev2 = 2;
151     {
152         auto dev = layer_mac.get_or_create_dev_data(dev1);
153         dev->unicast_address_add({{0x01,0x23,0x45,0x67,0x89,0xab}})
                ;
154 
155         auto v4 = layer_ipv4.get_or_create_dev_data(dev1);
156         v4->local_address_add({{10, 0, 1, 10}});
157 
158     }
159     {
160         auto dev = layer_mac.get_or_create_dev_data(dev2);
161         dev->unicast_address_add({{0x00,0x23,0x14,0x30,0xce,0x74}})
                ;
162 
163         auto v4 = layer_ipv4.get_or_create_dev_data(dev2);
164         v4->local_address_add({{10, 0, 2, 10}});
165 
166         auto neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,5}}));
167         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
168         neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,1}}));
169         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
170     }
171     layer_ipv4.get_routing_table().add_route({{{10,0,1,0}}, 24,
            true, {{0,0,0,0}}, dev1});
172     layer_ipv4.get_routing_table().add_route({{{10,0,2,0}}, 24,
            true, {{0,0,0,0}}, dev2});
173     layer_ipv4.get_routing_table().add_route({{{0,0,0,0}}, 0, false
            , {{10,0,2,1}}, dev2});
174 
175     // Load test data
176     std::vector<packet_ptr> packets;
177     if(!load_packets("test.pcap", packets)) {
178         return 1;
179     }
180     std::cerr << "Loaded " << packets.size() << " packets" << std::
            endl;
181 
182     run_benchmark(packets, layer_mac, dev1);
183 
184     return 0;
185 }
```

Listing B.3: bench/bench_loop

```
1 #!/bin/sh
2 
3 echo "" > results.txt
4 
```

```
for i in {1..100}
do
    ./bench >> results.txt
done
```

Listing B.4: bench/bootstrap_mean.r

```
# Constants
rounds = 100000 # Number of bootstrap rounds(Monte-carlo case resampling)

# Load numbers from results file. Each result should have a seperate line.
data <- scan('results.txt')

# Perform bootstrap resampling
means <- sapply(c(1:rounds), function(x) sample(data, length(data), replace=TRUE))

# Output mean and standard deviation
cat(sprintf("Mean: %f", mean(means)))
cat(sprintf("SD:   %f", sd(means)))

# Find confidence interval on the bootstrapped data
confi <- quantile(means, probs = c(0.025, 0.975));
cat(sprintf("95%% conf. interval: %f <= %f <= %f", confi[1], mean(means), confi[2]))
```

Listing B.5: bench/build_clang

```
#!/bin/sh
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
pushd $DIR
time clang++ --std=c++11 bench.cpp -I.. -lpcap -o bench -Wfatal-errors -g -pthread -O3 -lpthread -Wall -stdlib=libc++ -lc++abi
# time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-errors -g -pthread -O0 -lpthread -Wall -stdlib=libc++ -lc++abi
RETVAL=$?
popd
exit $RETVAL
```

Listing B.6: bench/build_gcc

```
#!/bin/sh
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
pushd $DIR
time g++ --std=c++11 bench.cpp -I.. -lpcap -o bench -Wfatal-errors -g -pthread -O3 -lpthread -Wall
# time g++ --std=c++11 bench.cpp -I.. -lpcap -o bench -Wfatal-errors -g -pthread -lpthread -Wall
popd
```

Listing B.7: bench/build_generator

```
#!/bin/sh
```

```
2 DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
3 pushd $DIR
4 time clang++ --std=c++11 generator.cpp -I.. -lpcap -o generator -
      Wfatal-errors -g -pthread -O3 -lpthread -Wall -stdlib=libc++ -
      lc++abi
5 # time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
      errors -g -pthread -O0 -lpthread -Wall -stdlib=libc++ -lc++abi
6 RETVAL=$?
7 popd
8 exit $RETVAL
```

Listing B.8: bench/generator.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <random>
4  #include <array>
5  #include <iterator>
6  #include <algorithm>
7
8  #include "trokis/support/byte_tools.hpp"
9
10 using namespace trokis::byte_tools;
11
12 constexpr std::size_t packet_count = 20e6;
13 constexpr std::size_t packet_size = 64;
14 constexpr std::size_t zerofill = packet_size - 6 * 2 - 2 - 20;
15
16 int main(int argc, char *argv[])
17 {
18     std::ranlux48_base prng;
19     prng.seed(std::random_device()());
20     std::uniform_int_distribution<> dist(0, 99);
21     auto next_rand = [&] () {
22         return dist(prng);
23     };
24
25     std::ofstream output("test.pcap", std::ios_base::trunc
26         | std::ios_base::out);
27     std::ostream_iterator<uint8_t> f(output);
28
29     // Fill in pcap header
30     write_bytes<endianness_host>((uint32_t)0xa1b2c3d4, f); // Magic
           number
31     write_bytes<endianness_host>((uint16_t)2, f);          // Major
           version
32     write_bytes<endianness_host>((uint16_t)4, f);          // Minor
           version
33     write_bytes<endianness_host>((int32_t)0, f);           //
         Timestamp in UTC
34     write_bytes<endianness_host>((uint32_t)0, f);          //
         Timestamp accurary
35     write_bytes<endianness_host>((uint32_t)~0, f);         //
         Snapshot length
```

```
36      write_bytes<endianness_host>((uint32_t)1, f);           //
            Ethernet link-layer type
37
38      // Create the base packet that we will modify and inject
            multiple times
39      std::array<uint8_t, packet_size> packet;
40      std::fill(std::begin(packet), std::end(packet), 0);
41
42      // Fill in ethernet header
43      write_bytes_array(std::array<uint8_t,6>({{0x1a,0x2b,0x3c,0x4d,0
            x5e,0x6f}}),
44          packet.begin() + 6);
45      write_bytes<endianness_big>((uint16_t)0x0800, packet.begin() +
            12);
46
47      // Fill in IP header
48      write_bytes<endianness_big>((uint8_t)0x45, packet.begin() + 14)
            ;
49      write_bytes<endianness_big>((uint16_t)50, packet.begin() + 16);
50      write_bytes<endianness_big>((uint8_t)64, packet.begin() + 22);
51      write_bytes<endianness_big>((uint8_t)17, packet.begin() + 23);
52      write_bytes_array(std::array<uint8_t,4>({{10,0,1,5}}), packet.
            begin() + 26);
53
54      // Fill the rest of the packet with UDP data
55      write_bytes<endianness_big>((uint16_t)0x7a69, packet.begin() +
            36);
56      write_bytes<endianness_big>((uint16_t)30, packet.begin() + 38);
57
58      for(std::size_t i = 0; i < packet_count; ++i) {
59          // Write pcap packet header
60          write_bytes<endianness_host>((uint32_t)0, f);
61          write_bytes<endianness_host>((uint32_t)0, f);
62          write_bytes<endianness_host>((uint32_t)packet_size, f);
63          write_bytes<endianness_host>((uint32_t)packet_size, f);
64
65          // Determine which kind of packet this will be
66          auto pkt_type = next_rand();
67
68          // Fill in dest MAC address
69          write_bytes_array(std::array<uint8_t,6>({{0x01,0x23,0x45,0
                x67,0x89,0xab}}), packet.begin() + 0);
70
71          if(pkt_type >= 55) {
72              // 45% to global IP(via nexthop)
73              write_bytes_array(std::array<uint8_t,4>({{1,2,3,4}}),
                    packet.begin() + 30);
74          } else if(pkt_type >= 10) {
75              // 45% to directly connected IP
76              write_bytes_array(std::array<uint8_t,4>({{10,0,2,5}}),
                    packet.begin() + 30);
77          } else if(pkt_type >= 5) {
78              // 5% Local received
79              write_bytes_array(std::array<uint8_t,4>({{10,0,1,10}}),
                     packet.begin() + 30);
```

```
80          } else {
81              // 5% Dropped at MAC layer
82              write_bytes_array(std::array<uint8_t,4>({{10,0,1,123}})
                    , packet.begin() + 30);
83              write_bytes_array(std::array<uint8_t,6>({{0xba,0x98,0
                    x76,0x54,0x32,0x10}}), packet.begin() + 0);
84          }
85          // Recalc the IP checksum
86          write_bytes<endianness_big>((uint16_t)0, packet.begin() +
                24);
87          auto csum = checksum_ones_complement(packet.begin() + 14,
                packet.begin() + 34);
88          write_bytes<endianness_big>((uint16_t)csum, packet.begin()
                + 24);
89
90          // Write ethernet header
91          std::copy(std::begin(packet), std::end(packet), f);
92      }
93
94      output.close();
95
96      return 0;
97  }
```

Listing B.9: functional_test/build_clang

```
1  #!/bin/sh
2  DIR="$(␣cd␣"$( dirname "${BASH_SOURCE[0]}" )"␣&&␣pwd␣)"
3  pushd $DIR
4  time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
       errors -g -pthread -O3 -lpthread -Wall -stdlib=libc++ -lc++abi
5  # time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
       errors -g -pthread -O0 -lpthread -Wall -stdlib=libc++ -lc++abi
6  RETVAL=$?
7  popd
8  exit $RETVAL
```

Listing B.10: functional_test/build_gcc

```
1  #!/bin/sh
2  DIR="$(␣cd␣"$( dirname "${BASH_SOURCE[0]}" )"␣&&␣pwd␣)"
3  pushd $DIR
4  time g++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-errors
       -g -pthread -O3 -lpthread -Wall
5  # time g++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
       errors -g -pthread -lpthread -Wall
6  popd
```

Listing B.11: functional_test/main.cpp

```
1  #include <iostream>
2  #include <cstdint>
3  #include <vector>
4  #include <array>
5  #include <algorithm>
```

```
6  #include <functional>
7  #include <chrono>
8  #include <unordered_map>
9  #include <map>
10 #include <tuple>
11 #include <atomic>
12 #include <fstream>
13
14 #include <iomanip>
15 #include <sstream>
16 #include <type_traits>
17 #include <iterator>
18
19 template<std::size_t N>
20 std::ostream& operator<<(std::ostream& os, const std::array<uint8_t
       , N>& array) {
21     bool first = true;
22     for(auto p : array) {
23         if(!first) {
24             os << ".";
25         }
26         os << (int)p;
27         first = false;
28     }
29     return os;
30 }
31
32
33 #include "trokis/types.hpp"
34 #include "trokis/packet_data.hpp"
35 #include "trokis/support/byte_tools.hpp"
36 #include "trokis/packet.hpp"
37 #include "trokis/support/dispatcher.hpp"
38 #include "trokis/protocols/ipv4/ipv4.hpp"
39 #include "trokis/protocols/ipv6/ipv6.hpp"
40 #include "trokis/protocols/ieee802/ieee802_3.hpp"
41 #include "trokis/protocols/ipv4/arp.hpp"
42
43 #include "trokis/support/pcap_output.hpp"
44
45 #include "trokis/devices/raw/netdevice_raw.hpp"
46
47 #include <pcap.h>
48
49 #include <signal.h>
50
51 using namespace trokis;
52
53 std::atomic<bool> running { true };
54
55 int main(int argc, char *argv[]) {
56     /*** Initialize network stack ***/
57     using namespace std::placeholders;
58
59     /* Set up MAC layer for ethernet */
```

```
60      ieee802::ethernet_802_3 layer_mac;
61
62      /* Set up receive path ethertype dispatch */
63      utilities::dispatcher<packet_verdict(packet_ptr&&, ieee802::
            ethertype, address_type)> dispatch;
64      dispatch.set_fallback([](packet_ptr&&, ieee802::ethertype,
            address_type) { return packet_verdict::dropped; });
65      layer_mac.set_hook_inform(std::ref(dispatch));
66      // layer_mac.set_hook_request(tools::pcap_output("ethernet.pcap
            "));
67
68      ipv4::ipv4_protocol layer_ipv4;
69
70      utilities::dispatcher<packet_verdict(packet_ptr&&, ipv4::
            protonum_t, ipv4::address_t, ipv4::address_t)>
            ipv4_dispatch;
71      ipv4_dispatch.set_fallback([](packet_ptr&&, ipv4::protonum_t,
            ipv4::address_t, ipv4::address_t) {
72          return packet_verdict::dropped;
73      });
74
75      layer_ipv4.set_hook_inform(std::ref(ipv4_dispatch));
76      neighbor::neighbor_table<neighbor::arp_provider<ipv4::address_t
            ,ieee802::dev_info_802_3>> neigh_ipv4_ethernet;
77      neigh_ipv4_ethernet.get_provider().set_output_hook(std::bind(&
            decltype(layer_mac)::transmit, &layer_mac, _1, _2, _3));
78      neigh_ipv4_ethernet.get_provider().set_primary_address_hook
            ([&](device_id devid) {
79          return std::make_pair(layer_ipv4.get_primary_address(devid)
                , layer_mac.get_primary_address(devid));
80      });
81      // Remember to switch to a demultiplexer here, if we start
            using lower level protocols that don't use MAC addresses.
82      layer_ipv4.set_hook_request(std::bind(&decltype(
            neigh_ipv4_ethernet)::transmit, &neigh_ipv4_ethernet, _1,
            _2));
83
84      /* Set up ARP receiver */
85      arp::arp_receiver layer_arp;
86      layer_arp.set_hook_request([&] (packet_ptr&& pkt, ieee802::
            mac_address src, ieee802::mac_address dst) {
87          pkt->data().push(2);
88          pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
89          return layer_mac.transmit(std::move(pkt), src, dst);
90      });
91      layer_arp.set_primary_address_hook(std::bind(&decltype(
            layer_mac)::get_primary_address, &layer_mac, _1));
92      layer_arp.set_is_local_address_hook(std::bind(&decltype(
            layer_ipv4)::is_local_address, &layer_ipv4, _1, _2));
93      layer_arp.set_update_neighbor_hook([&](device_id devid, ipv4::
            address_t upper, ieee802::mac_address lower, bool
            may_create) {
94          decltype(neigh_ipv4_ethernet.get(devid, upper)) neighbor;
95          if(may_create) {
```

```
96              neighbor = neigh_ipv4_ethernet.get_or_create(devid,
                    upper);
97          } else {
98              neighbor = neigh_ipv4_ethernet.get(devid, upper);
99          }
100         if(neighbor == nullptr) return false;
101         neighbor->set_lower_address(lower, ieee802::mac_address());
102         return true;
103     });
104
105     /* Set up ICMPv4 */
106     ipv4::icmp4::icmp4_protocol layer_ipv4_icmp;
107     layer_ipv4_icmp.set_hook_request(std::bind(&decltype(layer_ipv4
            )::transmit, &layer_ipv4, _1, _2, _3, _4));
108     ipv4_dispatch.set_handler(1, std::bind(&decltype(
            layer_ipv4_icmp)::receive, &layer_ipv4_icmp, _1, _2, _3));
109
110     /* Set up IPv6 */
111     ipv6::ipv6_protocol layer_ipv6;
112
113     // Set up dispatcher
114     dispatch.set_handler(0x0800, std::bind(&ipv4::ipv4_protocol::
            receive, &layer_ipv4, _1, _2));
115     dispatch.set_handler(0x86DD, std::bind(&ipv6::ipv6_protocol::
            receive, &layer_ipv6, _1, _2));
116     dispatch.set_handler(0x0806, std::bind(&arp::arp_receiver::
            receive, &layer_arp, _1, _2));
117
118     device_id dev1 = 1;
119     {
120         auto dev = layer_mac.get_or_create_dev_data(dev1);
121         dev->unicast_address_add({{0x00,0x23,0x14,0x30,0xce,0x74}})
                ;
122
123         auto v4 = layer_ipv4.get_or_create_dev_data(dev1);
124         v4->local_address_add({{10, 0, 0, 10}});
125
126         auto v6 = layer_ipv6.get_or_create_dev_data(dev1);
127         v6->local_address_add({{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 1}}, 0x2); // All nodes
128         v6->local_address_add({{0xfe, 0x80, 0, 0, 0, 0, 0, 0x02,
                0x23, 0x14, 0xff, 0xfe, 0x30, 0xce, 0x74}}, 0x2); //
                Link local
129     }
130
131     layer_ipv4.get_routing_table().add_route({{{10,0,0,0}}, 8, true
            , {{0,0,0,0}}, dev1});
132
133     // Set up a signal handler to catch Ctrl-C, so we can terminate
            correctly
134     struct sigaction signal_handler_INT;
135     signal_handler_INT.sa_handler = [](int s) { printf("Terminating
            ...\n"); running = false; };
136     sigemptyset(&signal_handler_INT.sa_mask);
137     signal_handler_INT.sa_flags = 0;
```

```
138     sigaction(SIGINT, &signal_handler_INT, nullptr);
139
140     devices::raw::netdevice_raw dev("side_b", 1024);
141     layer_mac.set_hook_request(std::bind(&decltype(dev)::send, &dev
            , _1));
142     auto worker_func = [&](int k) {
143         int x = 0;
144         while(running) {
145             int maxpull = 10;
146             auto retval = dev.poll(maxpull);
147             for(auto& pkt : retval) {
148                 --maxpull;
149                 ++x;
150                 pkt->dev = dev1;
151                 pkt->worker = k;
152                 layer_mac.receive(std::move(pkt));
153             }
154             if(maxpull != 0) {
155                 // Sleep if we did not receive a full batch of
                        packets.
156                 std::this_thread::sleep_for(std::chrono::
                        milliseconds(0));
157             }
158         }
159     };
160     std::vector<std::thread> workers;
161     for(auto i = 0; i < 2; ++i) {
162         workers.emplace_back(worker_func, i);
163     }
164     for(auto& w : workers) {
165         w.join();
166     }
167
168     return 0;
169 }
```

Listing B.12: trokis/containers/intrusive_queue_mpsc.hpp

```
1  //
2  //  intrusive_queue_mpsc.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 15/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_intrusive_queue_mpsc_hpp
10 #define trokis_intrusive_queue_mpsc_hpp
11
12 namespace trokis { namespace containers {
13
14     // Wait-free intrusive multiple-producer-single-consumer based
           on C version at:
15     //  http://www.1024cores.net/home/lock-free-algorithms/queues/
           intrusive-mpsc-node-based-queue
```

```
16      // Algorithm is unchanged from the C based version, but the
            interface is changed for better
17      // safety utilizing C++11 unique_ptr's. The given typename T
            must have at least a next member:
18      //  T* next
19      // The push interface accepts unique_ptr's and standard
            pointers. Only movable unique_ptr's are
20      // allowed, meaning the queue takes over ownership of the
            unique_ptr's object.
21      // The pop interface should only be called in single-threaded
            context and always returns
22      // unique_ptr's.
23      template<typename T>
24      class queue_mpsc {
25      public:
26          using value_type = T;
27
28          queue_mpsc() : head{&stub}, tail{&stub} {
29              stub.next = nullptr;
30          }
31
32          ~queue_mpsc() {
33              auto item = pop();
34              while(item) {
35                  item = pop();
36              }
37          }
38
39          // Takes control of a unique_ptr and adds the pointed to
                element to the list.
40          void push(std::unique_ptr<value_type>&& elem) {
41              auto tmp = std::move(elem);
42              push(tmp.release());
43          }
44
45          // Adds an element to the list in a wait-free manner.
46          void push(value_type* elem) {
47              if(elem == nullptr) {
48                  return;
49              }
50              elem->next = nullptr;
51              auto prev = head.exchange(elem);
52              // Note that a producer blocking right here will block
                    the consumer from seeing more items.
53              prev->next = elem;
54          }
55
56          // Returns the next item from the list.
57          std::unique_ptr<value_type> pop() {
58              auto tail_node = tail.load();
59              auto next_node = tail_node->next;
60              if(tail_node == &stub) {
61                  if(next_node == nullptr) {
62                      return nullptr;
63                  }
```

```
64                  tail.store(next_node);
65                  tail_node = next_node;
66                  next_node = next_node->next;
67              }
68              if(next_node != nullptr) {
69                  tail.store(next_node);
70                  return std::unique_ptr<value_type>(tail_node);
71              }
72              auto head_node = head.load();
73              if(tail_node != head_node) {
74                  return nullptr;
75              }
76              push(&stub);
77              next_node = tail_node->next;
78              if(next_node != nullptr) {
79                  tail.store(next_node);
80                  return std::unique_ptr<value_type>(tail_node);
81              }
82              return nullptr;
83          }
84
85      private:
86          value_type                 stub;
87          std::atomic<value_type*> head;
88          std::atomic<value_type*> tail;
89      };
90
91  } }
92
93  #endif
```

Listing B.13: trokis/detail/compat.h

```
1   //
2   //  compat.h
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 15/06/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_compat_h
10  #define trokis_compat_h
11
12  #include <cstddef>
13  #include <memory>
14  #include <type_traits>
15  #include <utility>
16
17  namespace std {
18      template<class T> struct _Never_true : false_type { };
19
20      template<class T> struct _Unique_if {
21          typedef unique_ptr<T> _Single;
22      };
```

```
23
24      template<class T> struct _Unique_if<T[]> {
25          typedef unique_ptr<T[]> _Runtime;
26      };
27
28      template<class T, size_t N> struct _Unique_if<T[N]> {
29          static_assert(_Never_true<T>::value, "make_unique forbids T
                [N]. Please use T[].");
30      };
31
32      template<class T, class... Args> typename _Unique_if<T>::
            _Single make_unique(Args&&... args) {
33          return unique_ptr<T>(new T(std::forward<Args>(args)...));
34      }
35
36      template<class T> typename _Unique_if<T>::_Single
            make_unique_default_init() {
37          return unique_ptr<T>(new T);
38      }
39
40      template<class T> typename _Unique_if<T>::_Runtime make_unique(
            size_t n) {
41          typedef typename remove_extent<T>::type U;
42          return unique_ptr<T>(new U[n]());
43      }
44
45      template<class T> typename _Unique_if<T>::_Runtime
            make_unique_default_init(size_t n) {
46          typedef typename remove_extent<T>::type U;
47          return unique_ptr<T>(new U[n]);
48      }
49
50      template<class T, class... Args> typename _Unique_if<T>::
            _Runtime make_unique_value_init(size_t n, Args&&... args) {
51          typedef typename remove_extent<T>::type U;
52          return unique_ptr<T>(new U[n]{ std::forward<Args>(args)...
                });
53      }
54
55      template<class T, class... Args> typename _Unique_if<T>::
            _Runtime make_unique_auto_size(Args&&... args) {
56          typedef typename remove_extent<T>::type U;
57          return unique_ptr<T>(new U[sizeof...(Args)]{ std::forward<
                Args>(args)... });
58      }
59  }
60
61  #endif
```

Listing B.14: trokis/devices/raw/netdevice_raw.hpp

```
1  //
2  //  netdevice_raw.hpp
3  //  trokis
4  //
```

```
5  //  Created by Lasse Bang Dalegaard on 18/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_netdevice_raw_hpp
10 #define trokis_netdevice_raw_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/support/dispatcher.hpp"
14
15 #include <string>
16 #include <vector>
17 #include <exception>
18 #include <cstring>
19 #include <fstream>
20
21 #include <sys/socket.h>
22 #include <sys/types.h>
23 #include <net/ethernet.h>
24 #include <net/if.h>
25 #include <netinet/in.h>
26 #include <linux/if_ether.h>
27 #include <netpacket/packet.h>
28 #include <unistd.h>
29 #include <fcntl.h>
30
31 namespace trokis { namespace devices { namespace raw {
32
33     namespace detail {
34
35         std::size_t interface_get_mtu(std::string devname) {
36             auto filename = std::string("/sys/class/net/") +
                    devname + std::string("/mtu");
37             std::ifstream input(filename);
38             std::size_t mtu;
39             input >> mtu;
40             return mtu;
41         }
42
43     }
44
45     // Implements a networking device that uses the Linux raw
            socket interface to send and receive
46     // raw ethernet frames to the network. This device is not multi
        -queue capable, but does support
47     // receiving from multiple concurrent threads, but data may be
            out of order in this case.
48     class netdevice_raw {
49     public:
50         netdevice_raw(std::string device_name, std::size_t backlog)
                : sock_(0) {
51             sock_ = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
52             if(sock_ == -1) {
53                 throw std::system_error(std::error_code(errno, std
                        ::system_category()), "Could not create socket"
```

```
                );
54          }
55
56          // Get interface MTU
57          mtu_ = detail::interface_get_mtu(device_name) + 14; //
                +14 bytes for ethernet header
58
59          // Find the interface index
60          ifindex_ = if_nametoindex(device_name.c_str());
61          if(ifindex_ == 0) {
62              throw std::runtime_error(std::string("Could not
                    find interface with name ") + device_name);
63          }
64
65          // Set up the sockaddr descriptor
66
67          sockaddr_ll lladdr;
68          memset(&lladdr, 0, sizeof(lladdr));
69          lladdr.sll_family = AF_PACKET;
70          lladdr.sll_protocol = htons(ETH_P_ALL);
71          lladdr.sll_ifindex = ifindex_;
72
73          // Bind to device
74          if(bind(sock_, reinterpret_cast<sockaddr*>(&lladdr),
                sizeof(lladdr)) != 0) {
75              throw std::system_error(std::error_code(errno, std
                    ::system_category()), "Could not bind to
                    interface");
76          }
77
78          // Set receive timeout
79
80          // Set non-blocking
81          if(fcntl(sock_, F_SETFL, O_NONBLOCK) != 0) {
82              throw std::system_error(std::error_code(errno, std
                    ::system_category()), "Could not set non-
                    blocking");
83          }
84
85          // Set options
86          packet_mreq req;
87          memset(&req, 0, sizeof(req));
88          req.mr_ifindex = ifindex_;
89          req.mr_type = PACKET_MR_PROMISC;
90          if(setsockopt(sock_, SOL_SOCKET, PACKET_ADD_MEMBERSHIP,
                &req, sizeof(req)) != 0) {
91              throw std::system_error(std::error_code(errno, std
                    ::system_category()), "Could not set promisc
                    mode");
92          }
93          memset(&req, 0, sizeof(req));
94          req.mr_ifindex = ifindex_;
95          req.mr_type = PACKET_MR_ALLMULTI;
96          if(setsockopt(sock_, SOL_SOCKET, PACKET_ADD_MEMBERSHIP,
                &req, sizeof(req)) != 0) {
```

```
97                  throw std::system_error(std::error_code(errno, std
                        ::system_category()), "Could not set all
                        multicast mode");
98              }
99          }
100         ~netdevice_raw() {
101             if(sock_ != 0) {
102                 close(sock_);
103             }
104         }
105
106         // Receives fully cooked ethernet packets from the upper
                layer protocols and
107         // forwards them on to the wire.
108         packet_verdict send(packet_ptr&& pkt) {
109             auto& pkt_data = pkt->data();
110             if(!pkt_data.may_pull(12)) return packet_verdict::
                    dropped;
111             auto segments = pkt_data.raw_segments();
112             std::vector<iovec> iov(segments.size());
113             transform(std::begin(segments), std::end(segments), std
                    ::begin(iov), [](const std::pair<byte_t*,std::
                    size_t>& v) {
114                 iovec retval;
115                 retval.iov_base = v.first;
116                 retval.iov_len = v.second;
117                 return retval;
118             });
119
120             sockaddr_ll lladdr;
121             memset(&lladdr, 0, sizeof(lladdr));
122             lladdr.sll_family = AF_PACKET;
123             lladdr.sll_halen = 6;
124             lladdr.sll_ifindex = ifindex_;
125             std::copy_n(pkt_data.begin() + 6, 6, lladdr.sll_addr);
126
127             msghdr msg;
128             memset(&msg, 0, sizeof(msg));
129             msg.msg_iov = iov.data();
130             msg.msg_iovlen = iov.size();
131             msg.msg_name = &lladdr;
132             msg.msg_namelen = sizeof(lladdr);
133             auto retval = sendmsg(sock_, &msg, 0);
134             if(retval < 0) {
135                 throw std::system_error(std::error_code(errno, std
                        ::system_category()), "Sending");
136             }
137             return (retval > 0) ? packet_verdict::success :
                    packet_verdict::dropped;
138         }
139
140         // Polls the ethernet device for packets and returns a
                vector of packets received.
141         std::vector<packet_ptr> poll(std::size_t max_packets) {
142             std::vector<packet_ptr> retval;
```

```
143             retval.reserve(max_packets);
144             for(std::size_t i = 0; i < max_packets; ++i) {
145                 // Allocate data
146                 std::vector<byte_t> buffer(mtu_);
147 retry:
148                 iovec iov;
149                 iov.iov_base = buffer.data();
150                 iov.iov_len  = buffer.size();
151                 sockaddr_ll lladdr;
152                 memset(&lladdr, 0, sizeof(lladdr));
153                 msghdr msg;
154                 memset(&msg, 0, sizeof(msg));
155                 msg.msg_iov = &iov;
156                 msg.msg_iovlen = 1;
157                 msg.msg_name = &lladdr;
158                 msg.msg_namelen = sizeof(lladdr);
159                 auto length = recvmsg(sock_, &msg, 0);
160                 if(length == 0) {
161                     // We will never receive a packet again
162                     break;
163                 }
164                 if(length < 0) {
165                     if(errno == EAGAIN || errno == EWOULDBLOCK) {
166                         // If the socket would have blocked, no
                                more packets can be pulled out. Return.
167                         break;
168                     }
169                     // Some unknown error message. We'll just throw
                            an exception.
170                     throw std::system_error(std::error_code(errno,
                            std::system_category()), "Error while
                            receiving");
171                 }
172                 if(lladdr.sll_pkttype == PACKET_OUTGOING) goto
                        retry;
173                 // A packet was received
174                 buffer.resize(length);
175                 auto data_ptr = std::make_unique<packet_data>(std::
                        move(buffer));
176                 auto ptr = std::make_unique<packet>();
177                 ptr->replace_data(std::move(data_ptr));
178                 retval.push_back(std::move(ptr));
179             }
180             return retval;
181         }
182
183     private:
184         std::size_t mtu_;
185         int sock_;
186         unsigned int ifindex_;
187     };
188
189 } } }
190
191 #endif
```

Listing B.15: trokis/neigh_provider/arp.hpp

```
1  //
2  //  arp_provider.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 15/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_arp_provider_hpp
10 #define trokis_arp_provider_hpp
11
12 #include "trokis/support/neighbor_table.hpp"
13
14 #include <functional>
15
16 namespace trokis { namespace neighbor {
17
18     // Implements ARP transmission, currently only for IP over
       ethernet but can of course be extended to any
19     // required protocol pair.
20     template<typename ProtocolType, typename HardwareInfo>
21     class arp_provider {
22     public:
23         using neighbor_type = neighbor::generic_neighbor<
           arp_provider>;
24         using upper_address_type = ProtocolType;
25         using lower_address_type = typename HardwareInfo::
           address_type;
26
27         using output_hook_type = std::function<packet_verdict(
           packet_ptr&&, lower_address_type, lower_address_type)>;
28         using primary_address_hook_type = std::function<std::pair<
           upper_address_type,lower_address_type>(device_id)>;
29
30         arp_provider()
31         : output_hook_([](packet_ptr&&, lower_address_type,
           lower_address_type) { return packet_verdict::dropped;
           })
32         , primary_address_hook_([](device_id) { return std::
           make_pair(upper_address_type(), lower_address_type());
           })
33         { }
34
35         template<typename Func>
36         void set_output_hook(Func&& f) {
37             output_hook_ = std::forward<Func>(f);
38         }
39
40         template<typename Func>
41         void set_primary_address_hook(Func&& f) {
42             primary_address_hook_ = std::forward<Func>(f);
43         }
44
```

```
45          // FUTURE: Update this so that other protocols are also
                supported, currently this only works if
46          // HardwareType = ieee802::mac_address and ProtocolType =
                ipv4::address (or equivalents)
47          void solicit(device_id devid, const upper_address_type&
                addr) {
48              // Create an ARP packet and send it
49              auto pkt = std::make_unique<packet>();
50              pkt->dev = devid;
51              auto& pkt_data = pkt->data();
52              pkt_data.reserve_front(64); // Reserve 64 bytes in
                    front to help out the lower levels.
53              pkt_data.put(28); // ARP with IP <-> ethernet is a 28
                    byte payload
54              pkt_data.set_bytes_be(+0, (uint16_t)1);          //
                    Ethernet has HTYPE = 1
55              pkt_data.set_bytes_be(+2, (uint16_t)0x0800);     // IP
                    has PTYPE = 0x0800, same as ethertype
56              pkt_data.set_byte(+4, 6);                        // 6
                    octets in a MAC address
57              pkt_data.set_byte(+5, 4);                        // 4
                    octets in an IP address
58              pkt_data.set_bytes_be(+6, (uint16_t)1);          // ARP
                    request has operation = 1
59
60              auto primary_addresses = primary_address_hook_(devid);
61
62              // Sender(that is our) address is the "primary" address
                     from the netdevice
63              pkt_data.set_bytes(+8, std::get<1>(primary_addresses));
64
65              // Our protocol address is our (primary) IP address
66              pkt_data.set_bytes(+14, std::get<0>(primary_addresses))
                    ;
67
68              // We don't know the target hardware address, so we
                    simply fill it with the ethernet broadcast address
69              pkt_data.set_bytes(+18, HardwareInfo::
                    addr_global_broadcast);
70
71              // Finally the IP we are looking for
72              pkt_data.set_bytes(+24, addr);
73
74              // Push an ethertype on, ARP is 0x0806
75              pkt->data().push(2);
76              pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
77
78              // Send it out. We don't care about the result of the
                    transmission itself.
79              raw_output(std::move(pkt), std::get<1>(
                    primary_addresses), HardwareInfo::
                    addr_global_broadcast);
80          }
81
82          // Performs raw output to the output hook.
```

```
83          packet_verdict raw_output(packet_ptr&& pkt, const
                lower_address_type& src, const lower_address_type& dst)
                 {
84              return output_hook_(std::move(pkt), src, dst);
85          }
86
87          // Encapsulates a frame and sends it to the output hook.
                The frame is encapsulated
88          // as an IPv4 frame, and assumes that the lower level is
                IEEE 802.3. Must be extended
89          // for full LLC support(and other protocols) later on.
90          packet_verdict output(packet_ptr&& pkt, const
                lower_address_type& src, const lower_address_type& dst)
                 {
91              pkt->data().push(2);
92              pkt->data().set_bytes_be(+0, (uint16_t)0x0800);
93              return raw_output(std::move(pkt), src, dst);
94          }
95
96      private:
97          output_hook_type output_hook_;
98          primary_address_hook_type primary_address_hook_;
99      };
100
101 } }
102
103 #endif
```

Listing B.16: trokis/packet.hpp

```
1  //
2  //  packet.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_packet_hpp
10 #define trokis_packet_hpp
11
12 #include "trokis/detail/compat.h"
13 #include "trokis/packet_data.hpp"
14
15 #include <memory>
16
17 namespace trokis {
18
19     // Wrapper around a packet. Basically a POD type containing
            high-level data about
20     // the packet. Also uniques a pointer to the actual payload
            data of the packet.
21     // The packet type performs no locking of any kind, as only a
            single thread is ever
```

```
22      // supposed to work with a given packet instance at any one
            time. In the client code,
23      // packet instances are always wrapped in packet_ptr objects.
24      class packet {
25      public:
26          packet() {
27              data_ = std::make_unique<packet_data>();
28          }
29
30          // Retrieves the payload data area
31          packet_data& data() const {
32              return *data_;
33          }
34
35          void replace_data(std::unique_ptr<packet_data>&& newdata) {
36              data_ = std::move(newdata);
37          }
38
39          // Device references
40          device_id dev; // "Current" device
41
42          // Worker that is handling this packet
43          worker_id worker;
44
45      private:
46          std::unique_ptr<packet_data> data_;
47      };
48
49  }
50
51  #endif
```

Listing B.17: trokis/packet_data.hpp

```
1   //
2   //  packet_data.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 09/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef TROKIS_PACKET_DATA
10  #define TROKIS_PACKET_DATA
11
12  #include "trokis/types.hpp"
13  #include "trokis/support/byte_tools.hpp"
14
15  #include <vector>
16  #include <cassert>
17
18  namespace trokis {
19
20      // Implements an example trokis packet_data backend. The
            packet_data structure
```

```
21      // implemenets a generic model for payload handling in the
            trokis network stack. Packet data
22      // is represented as an array of bytes with extra indexing to
            manage excess capacity.
23      // Conceptually, the data in a packet can be visualized as
            follows:
24      //
25      // (Begin)       Head                              Tail         (End)
26      //    |------------|--------------------------------|------------|
27      //    | Head room  |            Payload             | Tail room  |
28      //    |------------|--------------------------------|------------|
29      //
30      // This model is very similar to the end used in Linux, FreeBSD
            and others. From the users
31      // point of view, only the payload area is valid, and the head
            and tail rooms can be considered
32      // infinite. The API provides functionality for moving the head
            and tail pointers, and also
33      // provides functionality to read and write to the Payload area
            . All data access is relative to
34      // the Head pointer.
35      //
36      // Bytes can be read and written to using the get_* and set_
            functions, and direct access to the
37      // bytes in the payload area are available using the STL-like
            begin() and end() functions. These
38      // return iterators, allowing standard STL algorithms to be
            used with the container. The iterator
39      // returned by begin() points to the Head location while the
            iterator returned by end() points to
40      // the first location past Tail.
41      //
42      // The head pointer can be moved forward(payload size decreased
            ) using the pull(...) function,
43      // and backward(payload size increased) using the push(...)
            function. Both functions take the
44      // number of bytes to move the head pointer.
45      //
46      // Similarly, the tail pointer can be moved backward(payload
            size decreased) using trim(...),
47      // and forward(payload size increased) using put(...).
48      //
49      // Finally, head room can be reserved using the reserve_head
            function while the reserve_capacity
50      // function can be used to reserve a total length of the buffer
            . Space reservation should be
51      // considered a hint by the client.
52      //
53      // It is undefined if the iterators returned by begin() and end
            () are invalidated by Head/Tail
54      // movement, but the client should assume that they will be
            invalidated. Likewise, space reservation
55      // may invalidate all iterators.
56      //
```

```
57      // This initial implementation uses a simple std::vector as the
            backend buffer. This allows fast
58      // access to data because of the continuous layout in memory,
            but of course moving the Head or Tail
59      // pointer out of the current vector will result in
            reallocation of the memory buffer, forcing a
60      // full copy of all data. A future implementation could support
            fragmented data buffers, at the
61      // expense of somewhat more expensive get_* and set_* helpers(
            along with iterators) as these can no
62      // longer assume continous memory. The packet_data abstraction
            exists exactly to remove this detail
63      // from the client code.
64      //
65      // For transmitting, we provide the raw_segments function,
            allowing the client to get a list of raw
66      // data areas that should be combined to form the complete
            packet on outputting.
67      //
68      class packet_data {
69      public:
70          using backing_type = std::vector<byte_t>;
71
72          // In liu of inherited constructors, we can use this neat
                perfect forwarding trick
73          template<typename... Args>
74          packet_data(Args&&... args) : backing_(std::forward<Args>(
                args)...), head_(0) { }
75
76          packet_data() : backing_(), head_(0) { }
77
78          // Various attribute getters
79          std::size_t length() const {
80              return backing_.size() - head_;
81          }
82
83          std::size_t capacity() const {
84              return backing_.capacity();
85          }
86
87          // Reserve in front of head
88          void reserve_front(std::ptrdiff_t count) {
89              assert(backing_.size() >= head_);
90              if(length() == 0) {
91                  // No need to actually move data, but we need to
                        reserve at least this capacity
92                  backing_.resize(backing_.size() + count);
93              } else {
94                  backing_.insert(backing_.begin(), count, 0);
95              }
96              head_ += count;
97              assert(backing_.size() >= head_);
98          }
99
100         void reserve_length(std::ptrdiff_t count) {
```

```
101             backing_.reserve(head_ + count);
102         }
103
104         // Head adjustment
105         void push(std::size_t count) {
106             if(count > head_) {
107                 backing_.insert(backing_.begin(), count, 0);
108             } else {
109                 head_ -= count;
110             }
111         }
112
113         void pull(std::size_t count) {
114             head_ += count;
115             if(head_ > backing_.size()) {
116                 backing_.insert(backing_.end(), count, 0);
117             }
118         }
119
120         // Determine if we can read this amount of bytes
121         bool may_pull(std::size_t count) {
122             return length() >= count;
123         }
124
125         // Tail adjustment
126         void put(std::size_t count) {
127             backing_.resize(backing_.size() + count);
128         }
129         void trim(std::size_t count) {
130             backing_.resize(backing_.size() - count);
131             if(backing_.size() < head_) {
132                 head_ = backing_.size();
133             }
134         }
135
136         // Get iterators
137         backing_type::iterator begin() {
138             return backing_.begin() + head_;
139         }
140
141         backing_type::iterator end() {
142             return backing_.end();
143         }
144
145         backing_type::const_iterator cbegin() {
146             return backing_.cbegin() + head_;
147         }
148
149         backing_type::const_iterator cend() {
150             return backing_.cend();
151         }
152
153         // Retrieves bytes
154         template<std::size_t N>
155         uint64_t get_bytes_be(std::ptrdiff_t offset) const {
```

```
156             return byte_tools::read_bytes<N, byte_tools::
                    endianness_big>(backing_.data() + head_ + offset);
157         }
158
159         template<std::size_t N>
160         uint64_t get_bytes_le(std::ptrdiff_t offset) const {
161             return byte_tools::read_bytes<N, byte_tools::
                    endianness_little>(backing_.data() + head_ + offset
                    );
162         }
163
164         template<std::size_t N>
165         std::array<byte_t, N> get_bytes(std::ptrdiff_t offset)
                const {
166             return byte_tools::read_bytes_array<N>(backing_.data()
                    + head_ + offset);
167         }
168
169         byte_t get_byte(std::ptrdiff_t offset) const {
170             return *(backing_.data() + head_ + offset);
171         }
172
173         // Puts bytes
174         template<typename T>
175         void set_bytes_be(std::ptrdiff_t offset, T&& data) {
176             byte_tools::write_bytes<byte_tools::endianness_big>
177             (std::forward<T>(data), backing_.data() + head_ +
                    offset);
178         }
179
180         template<typename T>
181         void set_bytes_le(std::ptrdiff_t offset, T data) {
182             byte_tools::write_bytes<byte_tools::endianness_little>
183             (std::forward<T>(data), backing_.data() + head_ +
                    offset);
184         }
185
186         template<std::size_t N>
187         void set_bytes(std::ptrdiff_t offset, const std::array<
                byte_t, N>& data) {
188             byte_tools::write_bytes_array<N>(data, backing_.data()
                    + head_ + offset);
189         }
190
191         void set_byte(std::ptrdiff_t offset, byte_t data) {
192             *(backing_.data() + head_ + offset) = data;
193         }
194
195         // Returns the segments of the packet. Since this is backed
                by a vector, we simply return a
196         // one-element vector with a single (pointer,length) pair.
197         std::vector<std::pair<byte_t*, std::size_t>> raw_segments()
                {
198             return {std::make_pair(backing_.data() + head_, length
                    ())};
```

```
199         }
200
201     private:
202         backing_type backing_;
203         std::size_t head_;
204     };
205
206 }
207
208 #endif
```

Listing B.18: trokis/protocols/ieee802/ieee802_3.hpp

```
1  //
2  //  ieee802_3.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_802_hpp
10 #define trokis_802_hpp
11
12 #include "trokis/protocols/ieee802/types.hpp"
13
14 #include <mutex>
15 #include <unordered_map>
16 #include <cassert>
17
18 namespace trokis { namespace ieee802 {
19
20     namespace detail {
21         template<typename Func, typename... Args>
22         auto encap_ethertype(Func&& f, uint16_t ethertype,
                packet_ptr&& pkt, Args&&... args)
23         -> typename std::result_of<Func(packet_ptr&&, Args...)>::
                type {
24             pkt->data().push(2);
25             pkt->data().set_bytes_be(+0, ethertype);
26             return f(std::move(pkt), std::forward<Args>(args)...);
27         }
28     }
29
30     // Encapsulates upper 802 sublayers, namely Ethernet and LLC
            sublayers.
31     // These sublayers are in the same class because they need to
            call eachother and still have
32     // access to the hook to transmit to the upper level.
33     // MAC sublayers are allowed to inherit from this, giving them
            the required functionality to
34     // use in their own hooks if they require it.
35     template<typename DeviceData>
36     struct top_sublayers_802 : public utilities::generic_block<
37         packet_verdict(packet_ptr&&, ethertype, address_type),
```

```
38          packet_verdict(packet_ptr&& pkt)
39      > {
40
41          packet_verdict ether_decap_llc(packet_ptr&& pkt,
                address_type linklayer_type) {
42              // No LLC support yet.
43              return packet_verdict::dropped;
44          }
45
46          // Ethernet sublayer decapsulation
47          packet_verdict ether_decap(packet_ptr&& pkt, address_type
                linklayer_type) {
48              // Drop if we can't pull out an ethertype
49              if(!pkt->data().may_pull(2)) {
50                  return packet_verdict::dropped;
51              }
52              auto ethertype = pkt->data().get_bytes_be<2>(+0);
53              pkt->data().pull(2);
54              if(ethertype <= 1500) {
55                  // LLC sublayer packet
56                  return ether_decap_llc(std::move(pkt),
                        linklayer_type);
57              } else if(ethertype >= 1536) {
58                  // Ethernet sublayer packet. We don't yet support
                        VLANs, so simply inform upper.
59                  return inform(std::move(pkt), ethertype,
                        linklayer_type);
60              } else {
61                  return packet_verdict::dropped;
62              }
63          }
64
65          template<typename... Args>
66          DeviceData* get_or_create_dev_data(device_id id, Args&&...
                args) {
67              std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
68              auto retval = dev_data_.emplace(id, gc::make_gc_ptr<
                    DeviceData>(std::forward<Args>(args)...));
69              return retval.first->second.acquire();
70          }
71
72          void drop_dev_data(device_id id) {
73              std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
74              // Note that the actual dev_data entry will be
                    reclaimed later by gc_ptr
75              dev_data_.erase(id);
76          }
77
78          DeviceData* get_dev_data(device_id id) {
79              std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
80              auto iter = dev_data_.find(id);
81              if(iter == dev_data_.end()) return nullptr;
```

```
82              return iter->second.acquire();
83          }
84
85      private:
86          std::mutex dev_data_lock_;
87          std::unordered_map<device_id, gc::gc_ptr<DeviceData>>
                dev_data_;
88      };
89
90      // Encapsulates 802.3 MAC
91      class ethernet_802_3 : public top_sublayers_802<
            dev_data_ieee802_3> {
92      public:
93          // Receives 802.3 MAC frames and removes their initial MAC
                address header.
94          packet_verdict receive(packet_ptr&& pkt) {
95              auto dev = get_dev_data(pkt->dev);
96              if(!dev) {
97                  return packet_verdict::dropped;
98              }
99              // Drop if the packet isn't at least 12 bytes(dst + src
                    addresses)
100             if(!pkt->data().may_pull(12)) {
101                 return packet_verdict::dropped;
102             }
103
104             mac_address dst = pkt->data().get_bytes<6>(+0);
105             // Forward all multicast, all broadcast and only the
                    local unicast to upper layers.
106             auto is_multicast = (dst[0] & 0x1) != 0;
107             if(dev->has_unicast_address(dst) || is_multicast) {
108                 auto is_broadcast = (dst == dev_data_ieee802_3::
                        info::addr_global_broadcast);
109                 address_type type = is_multicast ? address_type::
                        multicast :
110                     (is_broadcast ? address_type::broadcast :
                            address_type::unicast);
111                 pkt->data().pull(12);
112                 return ether_decap(std::move(pkt), type);
113             } else {
114                 // FUTURE: Implement multicast and bridging
115                 return packet_verdict::dropped; // No bridging
                        support yet
116             }
117         }
118
119         mac_address get_primary_address(device_id devid) {
120             auto dev_data = get_dev_data(devid);
121             if(!dev_data) return dev_data_ieee802_3::info::
                    addr_global_broadcast;
122             return dev_data->get_primary_address();
123         }
124
125         // Packet transmission path. The packet must have already
                been protocol encapsulated
```

```
126         // at this point.
127         packet_verdict transmit(packet_ptr&& pkt, mac_address src,
                mac_address dst) {
128             auto dev = get_dev_data(pkt->dev);
129             if(!dev) {
130                 return packet_verdict::dropped;
131             }
132
133             if(src == mac_address()) {
134                 src = dev->get_primary_address();
135             }
136             pkt->data().push(12);
137             pkt->data().set_bytes(+0, dst);
138             pkt->data().set_bytes(+6, src);
139
140             // If the total length is less than 64, pad with zero.
141             int padlength = 64 - pkt->data().length();
142             if(padlength > 0) {
143                 pkt->data().put(padlength);
144                 std::fill(pkt->data().end() - padlength, pkt->data
                        ().end(), 0);
145             }
146
147             return request(std::move(pkt));
148         }
149     };
150
151 } }
152
153 #endif
```

Listing B.19: trokis/protocols/ieee802/types.hpp

```
1  //
2  //  ieee802_types.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 13/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ieee802_types_hpp
10 #define trokis_ieee802_types_hpp
11
12 #include "trokis/support/neighbor_table.hpp"
13 #include "trokis/neigh_provider/arp.hpp"
14
15 #include <array>
16
17 namespace trokis { namespace ieee802 {
18
19     // An IEEE 802 48-bit MAC address
20     // using mac_address = std::array<byte_t, 6>;
21
22     using mac_address = std::array<byte_t, 6>;
```

```
23
24      struct dev_info_802_3 {
25          static mac_address addr_global_broadcast;
26          using address_type = mac_address;
27      };
28      mac_address dev_info_802_3::addr_global_broadcast = {{0xFF, 0
            xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
29
30      // Ethertype field used in the Ethernet sublayer.
31      using ethertype = uint16_t;
32
33      // Describes an 802.3 networking device, ie. ethernet. Provides
             helper functions for managing
34      // addresses and configuration data of the device.
35      class dev_data_ieee802_3 {
36      public:
37          using info = dev_info_802_3;
38
39          dev_data_ieee802_3()
40          : unicast_addresses_(gc::make_gc_ptr<decltype(
                unicast_addresses_)::element_type>())
41          { }
42
43          // Helper: CAS updates the list of unicast addresses.
44          void unicast_address_add(mac_address addr) {
45              unicast_addresses_.update([&](decltype(
                    unicast_addresses_)::element_type& list) {
46                  list.push_back(addr);
47              });
48          }
49
50          // Helper: Tests if the interface has the specified address
                 as a unicast address
51          bool has_unicast_address(mac_address addr) {
52              auto& current_list = *unicast_addresses_;
53              auto retval = std::find(std::begin(current_list), std::
                    end(current_list), addr);
54              return retval != std::end(current_list);
55          }
56
57          // Helper: Retrieves the primary(first) address of the
                interface
58          mac_address get_primary_address() {
59              auto& current_list = *unicast_addresses_;
60              return current_list.size() != 0 ? *current_list.begin()
                     : info::addr_global_broadcast;
61          }
62
63      private:
64          gc::gc_ptr<std::vector<mac_address>> unicast_addresses_;
65      };
66
67  } }
68
69  #endif
```

Listing B.20: trokis/protocols/ipv4/arp.hpp

```
1  //
2  //  arp.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 14/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_arp_hpp
10 #define trokis_arp_hpp
11
12 #include "trokis/protocols/ipv4/types.hpp"
13 #include "trokis/protocols/ieee802/types.hpp"
14
15 namespace trokis { namespace arp {
16
17     // Receives and parses ARP packets. Only ethernet/IP is
       supported at the moment.
18     // Inputs:
19     //  packet_verdict receive(packet_ptr&&, mac_address,
       mac_address)
20     // Output hooks:
21     //  packet_verdict request(packet_ptr&&, mac_address)
22     // Data hooks:
23     //  bool update_neighbor(device_id, ip_address nexthop,
       mac_address newaddr, bool may_create)
24     //  bool is_local_address(device_id, ip_address address)
25     //
26     class arp_receiver : public utilities::
           generic_block_request_hook<
27         packet_verdict(packet_ptr&&, ieee802::mac_address, ieee802
               ::mac_address)
28     > {
29     public:
30         using update_neighbor_hook_t = std::function<bool(device_id
               , ipv4::address_t, ieee802::mac_address, bool)>;
31         using is_local_address_hook_t = std::function<bool(
               device_id, ipv4::address_t)>;
32         using primary_address_hook_t = std::function<ieee802::
               mac_address(device_id)>;
33
34         arp_receiver()
35         : update_neighbor_hook_([](device_id, ipv4::address_t,
               ieee802::mac_address, bool) { return false; })
36         , is_local_address_hook_([](device_id, ipv4::address_t) {
               return false; })
37         , primary_address_hook_([](device_id) { return ieee802::
               mac_address(); })
38         {};
39
40         packet_verdict receive(packet_ptr&& pkt, address_type
               linklayer_type) {
41             // If this device has no neighbor table, it shouldn't
                   support ARP so drop packet
```

```
42              if(pkt->data().length() < 28) { // Size of an ARP
                    packet with HTYPE = ethernet and PTYPE = IP
43                  return packet_verdict::dropped;
44              }
45              auto& pkt_data = pkt->data();
46
47              auto htype = pkt_data.get_bytes_be<2>(+0);
48              auto ptype = pkt_data.get_bytes_be<2>(+2);
49              if(htype != 1 || ptype != 0x0800) {
50                  // Ignore ARP packets that aren't for IP over
                        ethernet
51                  return packet_verdict::dropped;
52              }
53              if(pkt_data.get_byte(+4) != 6 || pkt_data.get_byte(+5)
                    != 4) {
54                  // Ignore ARP packet as the lengths don't match
55                  return packet_verdict::dropped;
56              }
57
58              // Read addresses
59              auto sha = pkt_data.get_bytes<6>(+8);   // Sender
                    Hardware Address
60              auto spa = pkt_data.get_bytes<4>(+14);  // Sender
                    Protocol Address
61              auto tpa = pkt_data.get_bytes<4>(+24);  // Target
                    Protocol Address
62
63              // Read the operation field
64              auto operation = pkt_data.get_bytes_be<2>(+6);
65
66              // Follow RFC826 packet reception section
67              bool merge_flag = false;
68              if(update_neighbor_hook_(pkt->dev, spa, sha, false)) {
69                  merge_flag = true;
70              }
71              if(is_local_address_hook_(pkt->dev, tpa)) {
72                  if(!merge_flag) {
73                      update_neighbor_hook_(pkt->dev, spa, sha, true)
                            ;
74                  }
75                  if(operation == 1) { // Message is a request and we
                        need to reply
76                      // Set the operation as REPLY = 2
77                      pkt_data.set_bytes_be(+6, (uint16_t)2);
78
79                      // Swap sender and target addresses
80                      std::swap_ranges(pkt_data.begin() + 8, pkt_data
                            .begin() + 18,
81                                       pkt_data.begin() + 18);
82
83                      // Fill in a new sender address(our own address
                            )
84                      auto host_address = primary_address_hook_(pkt->
                            dev);
85                      pkt_data.set_bytes(+8, host_address);
```

```
86                      return request(std::move(pkt), host_address,
                            sha);
87                  }
88              }
89              return packet_verdict::success;
90          }
91
92          template<typename Func>
93          void set_update_neighbor_hook(Func&& f) {
94              update_neighbor_hook_ = std::forward<Func>(f);
95          }
96
97          template<typename Func>
98          void set_is_local_address_hook(Func&& f) {
99              is_local_address_hook_ = std::forward<Func>(f);
100         }
101
102         template<typename Func>
103         void set_primary_address_hook(Func&& f) {
104             primary_address_hook_ = std::forward<Func>(f);
105         }
106
107     private:
108         update_neighbor_hook_t update_neighbor_hook_;
109         is_local_address_hook_t is_local_address_hook_;
110         primary_address_hook_t primary_address_hook_;
111     };
112
113 } }
114
115 #endif
```

Listing B.21: trokis/protocols/ipv4/defragmenter.hpp

```
1  //
2  //  defragmenter.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 17/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_defragmenter_hpp
10 #define trokis_defragmenter_hpp
11
12 #include "trokis/protocols/ipv4/types.hpp"
13
14 #include <tuple>
15 #include <chrono>
16 #include <mutex>
17 #include <vector>
18 #include <algorithm>
19
20 namespace trokis { namespace ipv4 {
21
```

```
22      // IPv4 packet defragmentation module
23      class defragmenter {
24      private:
25          // Tuple of src, dst, protocol and ID fields
26          using buffer_identifier = std::tuple<device_id, address_t,
                address_t, protonum_t, uint16_t>;
27          // Tuple of global offset, local offset, length and actual
                packet data to copy into the fragment. If
28          // packet_ptr is an empty pointer, this represents an empty
                span.
29          using fragment_area = std::tuple<std::size_t, std::size_t,
                std::size_t, packet_ptr>;
30
31          using fragment_clock = std::chrono::steady_clock;
32
33          // Holds a single fragment buffer. A list of packets is
                held, along with meta data to reassmeble the
34          // packet and also a timestamp to allow pruning the table
                later on.
35          struct fragment_buffer {
36              fragment_buffer()
37              : header_length(0), last_time(fragment_clock::now()),
                    total_length(0), buffered_count(0) { }
38
39              void reset() {
40                  header_length = 0;
41                  data_buffer.clear();
42                  total_length = 0;
43                  buffered_count = 0;
44              }
45
46              std::mutex lock;                                // Lock to
                    protect this fragment buffer
47              std::array<byte_t, 4 * 15>  header_buffer;  // Buffer
                    to contain header of final packet
48              std::size_t                 header_length;  // Length
                    of the header stored in header_buf
49              std::vector<fragment_area>  data_buffer;    // List of
                    fragment areas for the final fragment.
50                                                          // Note
                                                                that we
                                                                 can
                                                                use a
                                                                vector
                                                                since
                                                                we are
                                                                under
                                                                lock
51              fragment_clock::time_point  last_time;      // Time
                    when a fragment was last received
52              std::size_t                 total_length;   // Total
                    length of the packet
53              std::size_t                 buffered_count; // Number
                    of bytes of the total length that we have buffered
54          };
```

```
55
56      public:
57          defragmenter() { }
58          defragmenter(defragmenter&& other) {
59              std::unique_lock<decltype(frag_buffer_lock_)> guard(
                    other.frag_buffer_lock_);
60              frag_buffers_ = std::move(other.frag_buffers_);
61          }
62
63          // Receives a fragment of an IPv4 packet and performs all
                actions required to reassemble the packet.
64          // If the packet was reassembled, the resulting reassembled
                 packet is returned. If the packet could
65          // not be reassembled yet, an empty packet_ptr is returned.
66          packet_ptr fragment_receive(packet_ptr&& pkt, size_t
                header_length, protonum_t protocol,
67                                      address_t src, address_t dst) {
68              auto& pkt_data = pkt->data();
69              // Read packet identifier
70              auto pkt_id = pkt_data.get_bytes_be<2>(+4);
71
72              auto workerid = pkt->worker;
73              auto devid = pkt->dev;
74
75              // Determine if there are more fragments
76              auto pkt_flags = pkt_data.get_byte(+6);
77              bool more_fragments = pkt_flags & 0x20;
78
79              // Determine if this is the first fragment
80              auto fragment_offset = pkt_data.get_bytes_be<2>(+6) &
                    (0x1FFF);
81              bool is_first = fragment_offset == 0;
82
83              // This is not a fragmented packet!
84              if(is_first && !more_fragments) {
85                  return std::move(pkt);
86              }
87
88              auto offset = fragment_offset * 8;
89              // The packet already arrived trimmed for us, meaning
                    we can simply get the length from
90              // packet data and subtract the header_length we were
                    supplied.
91              auto payload_length = pkt_data.length() - header_length
                    ;
92
93              auto id = std::make_tuple(devid, src, dst, protocol,
                    pkt_id);
94              decltype(frag_buffers_)::iterator ibuffer;
95              fragment_buffer* buffer;
96              { // Critical section getting the fragment_buffer
97                  std::unique_lock<decltype(frag_buffer_lock_)> guard
                        (frag_buffer_lock_);
98                  ibuffer = frag_buffers_.find(id);
99                  if(ibuffer == frag_buffers_.end()) {
```

```
100                     auto retval = frag_buffers_.emplace(id, gc::
                            make_gc_ptr<fragment_buffer>());
101                     ibuffer = retval.first;
102                 }
103                 buffer = ibuffer->second.acquire();
104             }
105             { // Critical section performing the defragmentation
106                 std::unique_lock<decltype(fragment_buffer::lock)>
                        guard(buffer->lock);
107                 buffer->last_time = fragment_clock::now();
108                 // If this is the first packet, we have to set the
                        header
109                 if(more_fragments && is_first) {
110                     std::copy(pkt_data.begin(), pkt_data.begin() +
                            header_length, std::begin(buffer->
                            header_buffer));
111                     buffer->header_length = header_length;
112                 }
113
114                 // If this is the last fragment, we can set the
                        total length of the new packet.
115                 if(!more_fragments && !is_first) {
116                     buffer->total_length = offset + payload_length;
117                 }
118
119                 // Insert the new fragment
120                 buffer->data_buffer.emplace_back(offset,
                        header_length, payload_length, std::move(pkt));
121
122                 // Account for received data
123                 buffer->buffered_count += payload_length;
124                 if((buffer->buffered_count + buffer->header_length)
                        > 65535) {
125                     // If the amount of data received exceeds the
                            amount that can fit in an IPv4 packet,
126                     // cancel reassembly.
127                     buffer->reset();
128                     return nullptr;
129                 }
130
131                 if(buffer->total_length != 0 && buffer->
                        total_length <= buffer->buffered_count) {
132                     if(buffer->total_length < buffer->
                            buffered_count || buffer->header_length ==
                            0) {
133                         // Someone sent us nasty data. To be safe,
                                drop everything.
134                         buffer->reset();
135                         return nullptr;
136                     }
137                     // We have received the entire packet, so now
                            we can reassemble it!
138                     // First we sort by incresing packet offset.
139                     std::sort(std::begin(buffer->data_buffer), std
                            ::end(buffer->data_buffer),
```

```
140                     [](const fragment_area& a, const
                            fragment_area& b) {
141                         return std::get<0>(a) < std::get
                                <0>(b);
142                     });
143                 // FUTURE: Change this to reuse a packet or
                        even better, combine the old packets. For
                        now,
144                 // allocate an entirely new packet and fill it
                        in. Since this data is going up the stack,
145                 // we don't generally need to reserve head room
                         as the packets are being decapsulated. The
146                 // overhead of allocation an entirely new
                        packet should be small, and the code is
                        must
147                 // simpler.
148                 auto newpkt = std::make_unique<packet>();
149                 newpkt->worker = workerid;
150                 newpkt->dev = devid;
151                 auto& newpkt_data = newpkt->data();
152                 newpkt_data.put(buffer->header_length + buffer
                        ->total_length);
153                 std::copy(std::begin(buffer->header_buffer),
                        std::end(buffer->header_buffer),
154                           newpkt_data.begin());
155                 std::size_t expected_offset = buffer->
                        header_length;
156                 for(auto& fragment : buffer->data_buffer) {
157                     std::size_t global_offset = std::get<0>(
                            fragment) + buffer->header_length;
158                     if(global_offset != expected_offset) {
159                         // There's a hole or overlap in the
                                data, bail out.
160                         buffer->reset();
161                         return nullptr;
162                     }
163                     std::size_t length = std::get<2>(fragment);
164                     if((global_offset + length) > newpkt_data.
                            length()) {
165                         // Nasty data, bail out.
166                         buffer->reset();
167                         return nullptr;
168                     }
169                     expected_offset += length;
170
171                     std::size_t local_offset = std::get<1>(
                            fragment);
172                     packet_ptr ptr = std::move(std::get<3>(
                            fragment));
173
174                     std::copy(ptr->data().begin() +
                            local_offset,
175                               ptr->data().begin() +
                                    local_offset + length,
```

```
176                                 newpkt_data.begin() +
                                        global_offset);
177                     }
178                     auto final_length = buffer->total_length +
                            buffer->header_length;
179                     if(expected_offset != final_length) {
180                         // Data missing, bail out.
181                         buffer->reset();
182                         return nullptr;
183                     }
184
185                     // Note that at this point, the original header
                            will be wrong. We update the length but
186                     // no other fields, since the next processing
                            step might will most likely pull the header
187                     // off anyway.
188                     newpkt_data.set_bytes_be(+2, final_length);
189
190                     // Since the fragment was reassembled, we could
                            remove the buffer from the buffer map,
191                     // but this would require that we re-get the
                            top-level lock first, after dropping the
192                     // per-buffer lock. This would mean that we
                            need to block right here, possibly delaying
193                     // packet delivery further. Instead, we choose
                            simply reset the fragment buffer, reclaming
194                     // its resources. The resulting fragment buffer
                            will take up very little space, and will
195                     // be reclaimed next time the table is pruned.
196                     buffer->reset();
197
198                     // The resulting packet is now baked and ready
                            for returning
199                     return std::move(newpkt);
200                 }
201             }
202
203             return nullptr;
204         }
205
206         // Prunes the fragment buffers of old entries. Should be
                called periodically by the client software.
207         void prune_tables() {
208             std::unique_lock<decltype(frag_buffer_lock_)> guard(
                    frag_buffer_lock_);
209             // FUTURE: Make the cut-off timeout configurable
210             auto cutoff = fragment_clock::now() - std::chrono::
                    seconds(10);
211             // Note that because we are using gc ptrs, we can
                    remove them from the
212             // map without the underlying data going away, allowing
                    others to finish. We still need
213             // to lock when we check the timestamp however.
214             for(auto cur = frag_buffers_.cbegin(); cur !=
                    frag_buffers_.cend(); ) {
```

```
215                 std::unique_lock<decltype(fragment_buffer::lock)>
                        guard(cur->second->lock);
216                 if(cur->second->last_time < cutoff) {
217                     // Note that the gc pinters will make sure the
                            actual buffer sticks around for a bit
                            longer.
218                     frag_buffers_.erase(cur++);
219                 } else {
220                     ++cur;
221                 }
222             }
223         }
224
225     private:
226
227         // OPTIMIZATION: Change this to a data structure with finer
                grained locking. Seldom used path,
228         // so not a clear win.
229         std::mutex frag_buffer_lock_;
230         std::map<buffer_identifier, gc::gc_ptr<fragment_buffer>>
                frag_buffers_;
231     };
232
233 } }
234
235 #endif
```

Listing B.22: trokis/protocols/ipv4/ipv4.hpp

```
1  //
2  //  ipv4.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ipv4_hpp
10 #define trokis_ipv4_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/packet.hpp"
14 #include "trokis/support/dispatcher.hpp"
15 #include "trokis/protocols/ipv4/types.hpp"
16 #include "trokis/protocols/ipv4/defragmenter.hpp"
17 #include "trokis/protocols/ipv4/upper/icmp4.hpp"
18
19 #include <unordered_map>
20 #include <algorithm>
21 #include <tuple>
22 #include <list>
23 #include <mutex>
24
25 namespace trokis { namespace ipv4 {
26
```

```
27      //
28      // Implements IPv4 functionality, including fragmentation and
            adressing. Currently does not
29      // perform any handling of options or multicast(but there is no
            filtering of output traffic,
30      // meaning that this is a type 1 multicast implementation. It
            is also possible to receive
31      // multicast traffic, but no multicast routing is done.).
32      //
33      // We choose not to implement options, as they are very
            seldomly used in moderne applications.
34      // While RFC791 specifies that options *must* be implemented by
            all nodes, we feel that a
35      // solution of simply forwarding the packets verbatim will
            comply to RFC1812 section 5.3.13:
36      //  "Unrecognized IP options in forwarded packets MUST be
            passed through unchanged."
37      // While RFC1812 also states several options that are mandatory
            , namely Source Route,
38      // Record Route and Timestamp. In all three cases, RFC1812 also
            specifies that the software
39      // may provide configuration to turn off the intrepretation of
            the option. Ignoring IP options
40      // thus won't cause any actual harm to the network
            transmissions. For these reasons, our
41      // implementation simply forwards options verbatim, without
            interpreting them at all. The result
42      // of this choice is a simpler implementation.
43      //
44      class ipv4_protocol : public utilities::generic_block<
45          packet_verdict(packet_ptr&&, protonum_t, address_t,
                address_t),
46          packet_verdict(packet_ptr&&, address_t)
47      > {
48      public:
49          ipv4_protocol()
50          : may_forward_(false) {}
51
52          packet_verdict icmp_reply(packet_ptr&& pkt, uint8_t type,
                uint8_t code,
53                  address_t src, address_t dst, int header_length) {
54              auto& pkt_data = pkt->data();
55              // Trim back the original packet header
56              pkt_data.trim(pkt_data.length() - header_length - 8);
57
58              // Push the ICMP checksum on
59              pkt_data.push(8);
60              pkt_data.set_byte(+0, type);
61              pkt_data.set_byte(+1, code);
62              pkt_data.set_bytes_be(+2, (uint16_t)0);
63
64              // Calculate the checksum
65              auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.begin(), pkt_data.end());
66              pkt_data.set_bytes_be(+2, checksum);
```

```
67
68              return transmit(std::move(pkt), src, dst, 1);
69          }
70
71          // Performs the local input part of the receive path
72          packet_verdict receive_local(packet_ptr&& pkt, std::size_t
                header_length,
73                                       address_t src, address_t dst)
                                             {
74              auto protocol = pkt->data().get_byte(+9);
75              pkt = defragmenter_.fragment_receive(std::move(pkt),
                    header_length, protocol, src, dst);
76              if(!pkt) {
77                  return packet_verdict::success; // Packet was
                        consumed by the defragmenter
78              }
79              // Might need to read out new header length
80              header_length = (pkt->data().get_byte(+0) & 0xF) * 4;
81              if(header_length < 20) {
82                  return packet_verdict::dropped;
83              }
84              pkt->data().pull(header_length);
85              return inform(std::move(pkt), protocol, src, dst);
86          }
87
88          // Performs the routing part of the receive path
89          packet_verdict receive_route(packet_ptr&& pkt, std::size_t
                header_length,
90                                       address_t src, address_t dst,
                                             dev_data* device_data) {
91              auto& pkt_data = pkt->data();
92
93              // FUTURE: Implement IPv4 options. Currently we simply
                    forward them verbatim.
94
95              // Decrement packet TTL
96              auto ttl = pkt_data.get_byte(+8);
97              if(--ttl == 0) {
98                  icmp_reply(std::move(pkt), 11, 0, src, dst,
                        header_length);
99                  return packet_verdict::dropped;
100             }
101             pkt_data.set_byte(+8, ttl);
102
103             auto route = get_routing_table().snapshot()->
                    find_route_lpm(dst);
104             if(route == nullptr) {
105                 icmp_reply(std::move(pkt), 3, 0, src, dst,
                        header_length);
106                 return packet_verdict::dropped;
107             }
108
109             auto nexthop = route->direct ? dst : route->
                    nexthop_addr;
110
```

```
111             pkt->dev = route->dev;
112             return output_lower(std::move(pkt), header_length,
                    nexthop, device_data);
113         }
114
115         // Recalculates the checksum of an IP packet. The packet is
                assumed to have already been length checked.
116         void recalc_checksum(packet_data& pkt_data, std::size_t
                header_length) {
117             // OPTIMIZATION: Change this to use incremental
                    updating of checksum, per RFC1624, or even use a
118             // stateless offload on the NIC.
119             pkt_data.set_bytes_be(+10, uint16_t(0));
120             auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.cbegin(), pkt_data.cbegin() +
                    header_length);
121             pkt_data.set_bytes_be(+10, uint16_t(checksum));
122         }
123
124         // If required by the given interface metrics, fragment an
                IPv4 packet and return both parts.
125         // If fragmentation is not required, only a single packet
                is returned. The first returned packet
126         // fragment will always be compliant to the given metrics(
                unless the interface cannot possibly
127         // accomodate IPv4), while the second fragment may be
                larger than the given interface MTU. The
128         // caller should repeatedly call fragment_packet with the
                second packet of the last call, until
129         // only a single packet is returned. Does NOT fix packet
                checksums!
130         std::pair<packet_ptr, packet_ptr> fragment_packet(
                packet_ptr&& pkt, std::size_t header_length,
131                                                             std::
                                                                tuple
                                                                <std
                                                                ::
                                                                size_t
                                                                , std
                                                                ::
                                                                size_t
                                                                >&
                                                                metrics
                                                                )
132         {
133             std::size_t maximum_payload;
134             std::size_t front_reserve;
135             std::tie(maximum_payload, front_reserve) = metrics;
136
137             auto retval = std::make_pair<packet_ptr, packet_ptr>(
                    nullptr, nullptr);
138             auto& pkt_data = pkt->data();
139             if(maximum_payload < pkt_data.length()) {
140                 // Can we fragment this packet?
141                 auto pkt1_flags = pkt_data.get_byte(+6);
```

```
142                 if(pkt1_flags & 0x40) { // Mask 0100 0000, bit 1
                        from MSB is "Don't Fragment"
143                     return retval; // Return if we are not allowed
                            to fragment
144                 }
145
146                 // Interface can't support IPv4
147                 if(maximum_payload < (header_length + 8)) {
148                     return retval;
149                 }
150
151                 // The number of bytes we can put in the payload
152                 auto payload_length = maximum_payload -
                        header_length;
153                 auto split_at = payload_length + header_length;
154                 auto nfb = payload_length / 8; // Number of
                        Fragment Blocks
155                 auto excess_length = pkt_data.length() - split_at;
156
157                 // Perform the fragmentation split
158                 auto pkt2 = std::make_unique<packet>();
159                 pkt2->dev = pkt->dev;
160                 pkt2->worker = pkt->worker;
161                 auto& pkt2_data = pkt2->data();
162                 pkt2_data.reserve_front(front_reserve);
163                 pkt2_data.put(header_length + excess_length);
164                 // Copy data
165                 std::copy_n(pkt_data.begin() + split_at,
                        excess_length, pkt2_data.begin() +
                        header_length);
166                 // Copy header
167                 std::copy_n(pkt_data.begin(), header_length,
                        pkt2_data.begin());
168                 pkt_data.trim(excess_length);
169
170                 // Fix up headers in the two new packets
171                 // Set More Fragments on the _first_ packet, don't
                        change it on the second.
172                 pkt1_flags |= 0x20; // Mask 0010 0000, bit 2 from
                        MSB is "More Fragments"
173                 pkt_data.set_byte(+6, pkt1_flags);
174
175                 // Set Total Length fields of packets
176                 pkt_data.set_bytes_be(+2, (uint16_t)split_at);
177                 pkt2_data.set_bytes_be(+2, (uint16_t)(header_length
                        + excess_length));
178
179                 // Set Fragment Offset in _second_ packet to the
                        outer packet offset + nfb, don't change first.
180                 auto fragment_offset = pkt2_data.get_bytes_be
                        <2>(+6);
181                 // Preserve top three flag bits.
182                 fragment_offset = (0xE000 & fragment_offset) | (0
                        x1FFF & ((fragment_offset & (0x1FFF)) + nfb));
```

```
183                 pkt2_data.set_bytes_be(+6, (uint16_t)
                        fragment_offset);
184
185                 retval.second = std::move(pkt2);
186             }
187             retval.first = std::move(pkt);
188             return retval;
189         }
190
191         // Returns a tuple with element 0 indicating the maximum
                payload that should be sent and
192         // element 1 indicating the amount of room that should be
                reserved in front of the IP header.
193         std::tuple<std::size_t, std::size_t> get_packet_metrics(
                address_t nexthop, device_id dev)
194         {
195             // FUTURE: Make this use the neighbor infrastructure to
                    override interface metrics
196             return std::make_tuple(1500, 14);
197         }
198
199         packet_verdict output_lower_one(packet_ptr&& pkt, std::
                size_t header_length, address_t nexthop) {
200             recalc_checksum(pkt->data(), header_length); //
                    Recalculate checksum
201             return request(std::move(pkt), nexthop);      // Output
202         }
203
204         // Performs delivery to the lower layer. In case the MTU of
                 the device is too low, this can fail
205         // and this performs the required fragmentation.
206         packet_verdict output_lower(packet_ptr&& pkt, std::size_t
                header_length, address_t nexthop, dev_data* ddata) {
207             auto metrics = get_packet_metrics(nexthop, pkt->dev);
208             if(pkt->data().length() <= std::get<0>(metrics)) { //
                    Fast path: no fragmentation
209                 return output_lower_one(std::move(pkt),
                        header_length, nexthop);
210             }
211             auto pkts = fragment_packet(std::move(pkt),
                    header_length, metrics);
212             for(;;) {
213                 if(!pkts.first) {
214                     // Fragmentation failed because of a Dont
                            Fragment bit, so send ICMP to the sender
215                     // if not local. Currently we don't check for
                            this being to a local address.
216                     auto dst = pkt->data().get_bytes<4>(+12);
217                     auto src = ddata->get_primary_address();
218                     icmp_reply(std::move(pkt), 3, 4, src, dst,
                            header_length);
219                     return packet_verdict::dropped;
220                 }
221                 auto verdict = output_lower_one(std::move(pkts.
                        first), header_length, nexthop);
```

```
222                 if(verdict != packet_verdict::success) {
223                     // A packet failed on output, return verdict
                            and stop processing.
224                     return verdict;
225                 }
226
227                 // Are there more packets to process?
228                 if(pkts.second) {
229                     pkts = fragment_packet(std::move(pkts.second),
                            header_length, metrics);
230                 } else {
231                     // If not, break the loop
232                     break;
233                 }
234             }
235             return packet_verdict::success;
236         }
237
238         template<typename... Args>
239         dev_data* get_or_create_dev_data(device_id id, Args&&...
                args) {
240             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
241             auto retval = dev_data_.emplace(id, gc::make_gc_ptr<
                    dev_data>(std::forward<Args>(args)...));
242             return retval.first->second.acquire();
243         }
244
245         void drop_dev_data(device_id id) {
246             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
247             // Note that the actual dev_data entry will be
                    reclaimed later by gc_ptr
248             dev_data_.erase(id);
249         }
250
251         dev_data* get_device_data(device_id id) {
252             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
253             auto iter = dev_data_.find(id);
254             if(iter == dev_data_.end()) return nullptr;
255             return iter->second.acquire();
256         }
257
258         // IPv4 input path. Performs basic safety checks on the
                packet and forwards it to either
259         // the routing or the local input path. If the packet is in
                malformed, it will be dropped here.
260         packet_verdict receive(packet_ptr&& pkt, address_type
                linklayer_type) {
261             auto* device_data = get_device_data(pkt->dev);
262             if(device_data == nullptr) return packet_verdict::
                    dropped;
263
264             /* Parse IPv4 packet per RFC 791 sec. 3.1 */
```

```
265             auto& pkt_data = pkt->data();
266
267             // Minimum size of IPv4 header is 4*5 = 20 bytes, so
                    check that we may
268             // pull at least that. (RFC1812 sec 5.2.2)
269             if(!pkt_data.may_pull(20)) {
270                 return packet_verdict::dropped;
271             }
272
273             // First byte: 4 bits of version followed by 4 bits of
                    header length(RFC791 sec 3.1)
274             auto first_byte = pkt_data.get_byte(+0);
275             if((first_byte & 0xF0) != 0x40) {
276                 // Not an IPv4 packet
277                 return packet_verdict::dropped;
278             }
279
280             // Verify that packet header length is OK (RFC1812 sec
                    5.2.2)
281             uint8_t header_length = (first_byte & 0xF) * 4;
282             if(header_length < 20) {
283                 return packet_verdict::dropped;
284             }
285
286             // Verify packet length (RFC1812 sec 5.2.2)
287             uint16_t total_length = pkt_data.get_bytes_be<2>(+2);
288             if(total_length < header_length) {
289                 return packet_verdict::dropped;
290             }
291             if(!pkt_data.may_pull(total_length)) {
292                 return packet_verdict::dropped;
293             }
294             // Trim back end of packet. Note that we already
                    ensured that the packet was atleast total_length.
295             auto excess_length = pkt_data.length() - total_length;
296             if(excess_length > 0) {
297                 pkt_data.trim(excess_length);
298             }
299
300             // Verify checksum, must silently drop failing packets
                    (RFC1812 sec 4.2.2.5)
301             auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.cbegin(), pkt_data.cbegin() +
                    header_length);
302             if(checksum != 0) {
303                 return packet_verdict::dropped;
304             }
305
306             // Read addresses
307             auto src = pkt_data.get_bytes<4>(+12);
308             auto dst = pkt_data.get_bytes<4>(+16);
309
310             // Note: we do _not_ check TTL here, RFC1812 sec.
                    4.2.2.9 specifies that we must not inspect TTL
                    unless
```

```
311             // we are routing.
312             if(!device_data->is_local_address(dst) && dst !=
                    addr_global_broadcast) {
313                 // Perform routing. We don't modify the header if
                        we can avoid it, so don't pull from the packet.
314                 if(!may_forward_) return packet_verdict::dropped;
315                 return receive_route(std::move(pkt), header_length,
                        src, dst, device_data);
316             } else {
317                 return receive_local(std::move(pkt), header_length,
                        src, dst);
318             }
319         }
320
321         // Transmit path for upper level
322         packet_verdict transmit(packet_ptr&& pkt, address_t src,
                address_t dst, protonum_t proto) {
323             auto& data = pkt->data();
324             if(data.length() > (65535 - 20)) { // Maximum length of
                    payload is 2^16-1 - header length
325                 return packet_verdict::dropped;
326             }
327
328             auto route = get_routing_table().snapshot()->
                    find_route_lpm(dst);
329             if(route == nullptr) { // Can't find a matching route
330                 return packet_verdict::dropped;
331             }
332             auto* device_data = get_device_data(route->dev);
333             if(device_data == nullptr) return packet_verdict::
                    dropped;
334
335
336             // Append 20 bytes of IPv4 header and fill it in
337             data.push(20);
338             data.set_byte(+0, 0x45); // IP version 4 and a header
                    length of 20 bytes. Options are not supported.
339             data.set_byte(+1, 0x00); // DSCP and ECN are set to
                    zero
340             data.set_bytes_be(+2, (uint16_t)data.length());
341             data.set_bytes_be(+4, (uint16_t)(rand() & 0xFFFF)); //
                    Set the identification field to a random value
342             data.set_bytes_be(+6, (uint16_t)0); // FUTURE: Add a
                    way to set Dont Fragment here, for PMTU Discovery
343             data.set_byte(+8, 64); // Set TTL
344             data.set_byte(+9, (uint8_t)proto);
345             data.set_bytes(+12, src);
346             data.set_bytes(+16, dst);
347
348             auto nexthop = route->direct ? dst : route->
                    nexthop_addr;
349             pkt->dev = route->dev;
350             return output_lower(std::move(pkt), 20, nexthop,
                    device_data);
351         }
```

```
352
353         routing_table& get_routing_table() {
354             return routing_table_;
355         }
356
357         // Helper for external systems to get the primary address
                of a device. Mainly for ARP.
358         address_t get_primary_address(device_id dev) {
359             auto dev_data = get_device_data(dev);
360             if(dev_data == nullptr) return address_t();
361             return dev_data->get_primary_address();
362         }
363
364         bool is_local_address(device_id dev, address_t addr) {
365             auto dev_data = get_device_data(dev);
366             if(dev_data == nullptr) return false;
367             return dev_data->is_local_address(addr);
368         }
369
370         // Enable or disable gateway functionality
371         void set_gateway(bool flag) {
372             may_forward_ = flag;
373         }
374     private:
375         std::atomic<bool> may_forward_;
376
377         routing_table routing_table_;
378         defragmenter defragmenter_;
379
380         std::mutex dev_data_lock_;
381         std::unordered_map<device_id, gc::gc_ptr<dev_data>>
                dev_data_;
382     };
383
384 } }
385
386 #endif
```

Listing B.23: trokis/protocols/ipv4/types.hpp

```
1  //
2  //  ipv4_types.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 12/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ipv4_types_hpp
10 #define trokis_ipv4_types_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/support/gc.hpp"
14 #include "trokis/support/routing_table.hpp"
15
```

```
16 #include <array>
17 #include <iterator>
18 #include <algorithm>
19 #include <cassert>
20
21 namespace trokis { namespace ipv4 {
22
23     using address_t = std::array<byte_t, 4>;
24     using protonum_t = uint8_t;
25
26     static constexpr address_t addr_global_broadcast {{255, 255,
           255, 255}};
27
28     class routing_table {
29     public:
30         routing_table()
31         : routing_table_(gc::make_gc_ptr<decltype(routing_table_)::
               element_type>()) { }
32
33         routing::routing_table<address_t>* snapshot() const {
34             return routing_table_.acquire();
35         }
36
37         void add_route(const routing::route<address_t>& route) {
38             routing_table_.update([&](decltype(routing_table_)::
                   element_type& table) {
39                 table.add_route(route);
40             });
41         }
42
43     private:
44         gc::gc_ptr<routing::routing_table<address_t>>
               routing_table_;
45     };
46
47     class dev_data {
48     public:
49         dev_data() : unicast_addresses_(gc::make_gc_ptr<decltype(
               unicast_addresses_)::element_type>())
50         { }
51
52         bool is_local_address(address_t addr) {
53             auto& current_list = *unicast_addresses_;
54             auto retval = std::find(std::begin(current_list), std::
                   end(current_list), addr);
55             return retval != std::end(current_list);
56         }
57
58         void local_address_add(address_t addr) {
59             unicast_addresses_.update([&](decltype(
                   unicast_addresses_)::element_type& list) {
60                 list.push_back(addr);
61             });
62         }
63
```

```
64          address_t get_primary_address() {
65              auto& current_list = *unicast_addresses_;
66              return current_list.size() != 0 ? *current_list.begin()
                    : address_t();
67          }
68
69      private:
70          gc::gc_ptr<std::vector<address_t>> unicast_addresses_;
71      };
72
73  } }
74
75  #endif
```

Listing B.24: trokis/protocols/ipv4/upper/icmp4.hpp

```
1   //
2   //  icmp4.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 17/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_icmp4_hpp
10  #define trokis_icmp4_hpp
11
12  namespace trokis { namespace ipv4 { namespace icmp4 {
13
14      // Implements basic parts of RFC792
15      class icmp4_protocol
16      : public utilities::generic_block_request_hook<
17        packet_verdict(packet_ptr&& pkt, address_t dst, address_t src
              , protonum_t proto)
18      > {
19      public:
20
21          // Receives an ICMPv4 packet and performs required parsing.
22          packet_verdict receive(packet_ptr&& pkt, address_t src,
                address_t dst) {
23              auto& pkt_data = pkt->data();
24              if(!pkt_data.may_pull(28)) { // Length of ICMP header +
                     shortest IP header
25                  return packet_verdict::dropped;
26              }
27
28              auto type = pkt_data.get_byte(+0);
29              auto code = pkt_data.get_byte(+1);
30              // auto checksum = pkt_data.get_bytes_be<2>(+2);
31              pkt_data.pull(4);
32              switch(type) {
33              case 8:
34                  if(code != 0) return packet_verdict::dropped;
35                  return receive_ping(std::move(pkt), src, dst);
36                  break;
```

```
37              default:
38                  return packet_verdict::dropped;
39                  break;
40              }
41          }
42
43      private:
44          void encap_icmp(packet_data& pkt_data, uint8_t type,
                uint8_t code) {
45              pkt_data.push(4);
46              pkt_data.set_byte(+0, type);
47              pkt_data.set_byte(+1, code);
48          }
49
50          void fix_checksum(packet_data& pkt_data) {
51              pkt_data.set_bytes_be(+2, (uint16_t)0);
52              auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.begin(), pkt_data.end());
53              pkt_data.set_bytes_be(+2, (uint16_t)checksum);
54          }
55
56          packet_verdict receive_ping(packet_ptr&& pkt, address_t src
                , address_t dst) {
57              // In this case we modify the original packet and send
                    it back out.
58              encap_icmp(pkt->data(), 0, 0);
59              fix_checksum(pkt->data());
60              return request(std::move(pkt), dst, src, 1);
61          }
62      };
63
64
65  } } }
66
67  #endif
```

Listing B.25: trokis/protocols/ipv6/ipv6.hpp

```
1   //
2   //  ipv6.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 09/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_ipv6_hpp
10  #define trokis_ipv6_hpp
11
12  #include "trokis/types.hpp"
13  #include "trokis/protocols/ipv6/types.hpp"
14
15  namespace trokis { namespace ipv6 {
16
17      //
```

```
18      // Implements the IP Layer of the IPv6 protocol(RFC 2460) as
            required by RFC 4294.
19      //
20      // Our implementation ignores the Flow Label field as specified
            by RFC 2460, forwarding
21      // it unchanged, originating packets with it set to zero and
            ignoring it on reception,
22      // as allowed by RFC 2460 section 6.
23      //
24      // Our implementation reads the traffic class field and
            forwards it to the upper layer
25      // protocols, but it does not make any attempt to decode the
            meaning of the field. We
26      // also allow upper layer protocols to supply a value to the
            field.
27      //
28      // Unlike the IPv4 implementation, we do not implement any IPv6
            options here. Instead,
29      // we let upper level handlers manage these. This is made
            possible by the fact that IPv6
30      // does not use an embedded list of options but instead uses a
            linked list of headers
31      // with every header containing the type of the next header in
            the list. When forwarding
32      // is implemented, the Hop-by-Hop option should be managed in
            in this module.
33      //
34      class ipv6_protocol : utilities::generic_block<
35          packet_verdict(packet_ptr&&, protonum_t, address_t,
                address_t, byte_t),
36          packet_verdict(packet_ptr&&, address_t)
37      >{
38      public:
39          packet_verdict receive_local(packet_ptr&& pkt, address_t
                dst) {
40              auto& pkt_data = pkt->data();
41
42              // Perform options parsing
43              auto next_header = pkt_data.get_byte(+6);
44              if(next_header == 59) return packet_verdict::success;
45
46              // Get the data we need to pass to upper layer
47              auto src = pkt_data.get_bytes<16>(+8);
48              auto tc = (pkt_data.get_bytes_be<2>(+0) >> 4) & 0xFF;
49              pkt_data.pull(40); // Pull off the IPv6 header
50
51              return inform(std::move(pkt), next_header, src, dst, tc
                    );
52          }
53
54          packet_verdict receive(packet_ptr&& pkt, address_type
                linklayer_type) {
55              auto dev_data = get_device_data(pkt->dev);
56              if(dev_data == nullptr) {
57                  return packet_verdict::dropped;
```

```
58              }
59
60              auto& pkt_data = pkt->data();
61
62              // Verify that the packet can fit at least the IPv6
                    header
63              if(!pkt_data.may_pull(40)) {
64                  return packet_verdict::dropped;
65              }
66
67              // Read version and verify it
68              if((pkt_data.get_byte(+0) >> 4) != 6) {
69                  return packet_verdict::dropped; // Not IPv6
70              }
71
72              // Get payload length
73              auto payload_length = pkt_data.get_bytes_be<2>(+4);
74              auto total_length = payload_length + 40;
75              if(!pkt_data.may_pull(total_length)) {
76                  return packet_verdict::dropped;
77              }
78              // Trim packet to correct length
79              pkt_data.trim(pkt_data.length() - total_length);
80
81              // Get destination address
82              auto dst = pkt_data.get_bytes<16>(+24);
83
84              if(dev_data->is_local_address(dst)) {
85                  return receive_local(std::move(pkt), dst);
86              } else {
87                  return packet_verdict::dropped;
88              }
89          }
90
91          packet_verdict transmit(packet_ptr&& pkt, address_t dst,
                address_t src,
92                  byte_t traffic_class, protonum_t next_header) {
93              auto& pkt_data = pkt->data();
94
95              auto length = pkt_data.length();
96              if(length > 0xFFFF) {
97                  return packet_verdict::dropped;
98              }
99
100             auto route = get_routing_table().snapshot()->
                    find_route_lpm(dst);
101             if(route == nullptr) { // Can't find a matching route
102                 return packet_verdict::dropped;
103             }
104             auto dev_data = get_device_data(route->dev);
105             if(dev_data == nullptr) {
106                 return packet_verdict::dropped;
107             }
108
109             // Insert IPv6 header
```

```
110             pkt_data.push(40);
111             // Version, traffic class and flow label(= 0)
112             uint32_t first_word = (4 << 28) | (traffic_class << 20)
                    ;
113             pkt_data.set_bytes_be(+0, first_word);
114
115             // Payload length
116             pkt_data.set_bytes_be(+4, (uint16_t)length);
117             pkt_data.set_byte(+6, next_header);
118             pkt_data.set_byte(+7, 64);
119             pkt_data.set_bytes(+8, src);
120             pkt_data.set_bytes(+24, dst);
121
122             auto nexthop = route->direct ? dst : route->
                    nexthop_addr;
123             pkt->dev = route->dev;
124             return request(std::move(pkt), nexthop);
125         }
126
127         routing_table& get_routing_table() {
128             return routing_table_;
129         }
130
131         template<typename... Args>
132         dev_data* get_or_create_dev_data(device_id id, Args&&...
                args) {
133             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
134             auto retval = dev_data_.emplace(id, gc::make_gc_ptr<
                    dev_data>(std::forward<Args>(args)...));
135             return retval.first->second.acquire();
136         }
137
138         void drop_dev_data(device_id id) {
139             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
140             // Note that the actual dev_data entry will be
                    reclaimed later by gc_ptr
141             dev_data_.erase(id);
142         }
143
144         dev_data* get_device_data(device_id id) {
145             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
146             auto iter = dev_data_.find(id);
147             if(iter == dev_data_.end()) return nullptr;
148             return iter->second.acquire();
149         }
150
151     private:
152         routing_table routing_table_;
153
154         std::mutex dev_data_lock_;
155         std::unordered_map<device_id, gc::gc_ptr<dev_data>>
                dev_data_;
```

```
156     };
157
158 } }
159
160 #endif
```

Listing B.26: trokis/protocols/ipv6/types.hpp

```
1  //
2  //  ipv6_types.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 12/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ipv6_types_hpp
10 #define trokis_ipv6_types_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/support/routing_table.hpp"
14
15 namespace trokis { namespace ipv6 {
16
17     using address_t = std::array<byte_t, 16>;
18     using protonum_t = byte_t;
19     using scope_t = byte_t;
20
21     class routing_table {
22     public:
23         routing_table()
24         : routing_table_(gc::make_gc_ptr<decltype(routing_table_)::
               element_type>()) { }
25
26         routing::routing_table<address_t>* snapshot() const {
27             return routing_table_.acquire();
28         }
29
30         void add_route(const routing::route<address_t>& route) {
31             routing_table_.update([&](decltype(routing_table_)::
                   element_type& table) {
32                 table.add_route(route);
33             });
34         }
35
36     private:
37         gc::gc_ptr<routing::routing_table<address_t>>
               routing_table_;
38     };
39
40     // Fat POD type for containing data describing an interface.
           Helpers are provided for all data
41     // that needs locking.
42     class dev_data {
43     public:
```

```
44          using address_desc_type = std::tuple<address_t,scope_t>;
45
46          dev_data() : local_addresses_(gc::make_gc_ptr<decltype(
                local_addresses_)::element_type>())
47          { }
48
49          bool is_local_address(address_t addr) {
50              auto& current_list = *local_addresses_;
51              auto retval = std::find_if(std::begin(current_list),
                    std::end(current_list),
52                  [&] (const std::tuple<address_t,scope_t>& val) {
                        return std::get<0>(val) == addr; });
53              return retval != std::end(current_list);
54          }
55
56          void local_address_add(address_t addr, scope_t scope) {
57              local_addresses_.update([&](decltype(local_addresses_)
                    ::element_type& list) {
58                  list.emplace_back(addr, scope);
59              });
60          }
61
62          void local_addresses_remove(address_t addr) {
63              local_addresses_.update([&](decltype(local_addresses_)
                    ::element_type& list) {
64                  std::remove_if(std::begin(list), std::end(list),
65                      [&] (const std::tuple<address_t,scope_t>& val)
                            { return std::get<0>(val) == addr; });
66              });
67          }
68
69          std::vector<address_desc_type> get_local_addresses() {
70              auto& current_list = *local_addresses_;
71              return std::vector<address_desc_type>(std::begin(
                    current_list), std::end(current_list));
72          }
73
74          void set_iface_identifier(std::array<byte_t, 8> id) {
75              iface_identifier_ = id;
76          }
77
78          std::array<byte_t, 8> iface_identifier() {
79              return iface_identifier_;
80          }
81
82      private:
83          gc::gc_ptr<std::vector<address_desc_type>> local_addresses_
                ;
84          std::array<byte_t, 8> iface_identifier_;
85      };
86
87  } }
88
89  #endif
```

Listing B.27: trokis/support/byte_swappers.hpp

```
//
//  byte_swappers.hpp
//  trokis
//
//  Created by Lasse Bang Dalegaard on 12/07/13.
//  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
//

#ifndef trokis_byte_swappers_hpp
#define trokis_byte_swappers_hpp

#include "trokis/types.hpp"

#include <algorithm>

namespace trokis { namespace byte_tools {

    // Generic byte-swapping mechanism based on std::reverse. Reverses
    // a byte array, in effect performing a byte swap.
    template<std::size_t N>
    struct byte_swapper_reverse {
        inline static void swap(std::array<byte_t, N>& raw) {
            std::reverse(raw.begin(), raw.begin() + N);
        }
    };

} }

#endif
```

Listing B.28: trokis/support/byte_tools.hpp

```
//
//  byte_tools.hpp
//  trokis
//
//  Created by Lasse Bang Dalegaard on 09/07/13.
//  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
//

#ifndef trokis_byte_tools_hpp
#define trokis_byte_tools_hpp

#include <boost/detail/endian.hpp>
#include "trokis/support/byte_swappers.hpp"

#include <algorithm>

namespace trokis { namespace byte_tools {

    struct endianness_little;
    struct endianness_big;

```

```
22 #if defined(BOOST_LITTLE_ENDIAN)
23     using endianness_host = endianness_little;
24 #elif defined(BOOST_BIG_ENDIAN)
25     using endianness_host = endianness_big;
26 #else
27 #error "System endianness unknown"
28 #endif
29
30     // Identity byte swapper(ie. does nothing)
31     template<std::size_t N>
32     struct byte_swapper_id {
33         static void swap(std::array<byte_t, N>& raw) { }
34     };
35
36     // Generic byte swapping interface
37     template<std::size_t N, typename SourceEndianness, typename
           DestinationEndianness>
38     struct byte_swapper;
39
40     // Byte swapping specializations to/from little/endian. Uses
           reversing or identity.
41     template<std::size_t N>
42     struct byte_swapper<N, endianness_little, endianness_little> :
           byte_swapper_id<N> { };
43     template<std::size_t N>
44     struct byte_swapper<N, endianness_little, endianness_big> :
           byte_swapper_reverse<N> { };
45     template<std::size_t N>
46     struct byte_swapper<N, endianness_big, endianness_little> :
           byte_swapper_reverse<N> { };
47     template<std::size_t N>
48     struct byte_swapper<N, endianness_big, endianness_big> :
           byte_swapper_id<N> { };
49
50     template<std::size_t N>
51     struct nbyte_type {
52         using type = uint64_t;
53     };
54
55     template<>
56     struct nbyte_type<2> {
57         using type = uint16_t;
58     };
59
60     template<std::size_t N>
61     using nbyte_type_t = typename nbyte_type<N>::type;
62
63     // Directly reads bytes, no swapping of endianness or similar
           is performed
64     template<std::size_t N, typename ForwardIt>
65     std::array<uint8_t, N> read_bytes_array(const ForwardIt& begin)
           {
66         std::array<byte_t, N> tmp;
67         std::copy(begin, begin + N, tmp.begin());
68         return tmp;
```

```
69      }
70
71      // Generic read_bytes call taking the forward input iterator
            for a container. Does _not_ perform bounds checking.
72      template<std::size_t N, typename Endianness, typename ForwardIt
            >
73      nbyte_type_t<N> read_bytes(ForwardIt&& begin) {
74          auto tmp = read_bytes_array<N>(std::forward<ForwardIt>(
                begin));
75          asm("#preswap");
76          byte_swapper<N, Endianness, endianness_host>::swap(tmp);
77          asm("#postswap");
78          return *((nbyte_type_t<N>*)tmp.data());
79      }
80
81      // Directly writes bytes, no swapping of endianness or similar
            is performed
82      template<std::size_t N, typename ForwardIt>
83      void write_bytes_array(const std::array<byte_t, N>& data, const
             ForwardIt& begin) {
84          std::copy(data.begin(), data.end(), begin);
85      }
86
87
88      template<typename T>
89      struct convert_to_bytes_impl {
90          static constexpr std::size_t length = sizeof(T);
91
92          static std::array<byte_t, length> convert(T val) {
93              std::array<byte_t, length> tmp;
94              byte_t* raw = reinterpret_cast<byte_t*>(&val);
95              std::copy(raw, raw + length, tmp.begin());
96              return tmp;
97          }
98      };
99
100     // Generic write_bytes calls. Takes data and forward output
            iterator
101     template<typename Endianness, typename T, typename ForwardIt>
102     void write_bytes(T&& data, ForwardIt&& begin) {
103         using converter = convert_to_bytes_impl<T>;
104         auto raw = converter::convert(std::forward<T>(data));
105         byte_swapper<converter::length, endianness_host, Endianness
                >::swap(raw);
106         write_bytes_array(raw, std::forward<ForwardIt>(begin));
107     }
108
109     // Performs a ones-complement sum of all 2-octet words in the
            given iterator range. Used by IP, UDP and TCP
110     // for checksum calculations. See RFC 1071.
111     template<typename ForwardIt>
112     uint16_t checksum_ones_complement(const ForwardIt& begin, const
             ForwardIt& end) {
113         uint32_t csum = 0;
114         bool shift = true;
```

```
115         for(auto cur = begin; cur != end; ++cur) {
116             csum += *cur << (shift ? 8 : 0);
117             shift = !shift;
118         }
119         while(csum > 0xFFFF) {
120             csum = (csum & 0xFFFF) + (csum >> 16);
121         }
122         return ~(csum) & 0xFFFF;
123     }
124
125 } }
126
127 #endif
```

Listing B.29: trokis/support/dispatcher.hpp

```
1  //
2  //  dispatcher_func.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_dispatcher_func_hpp
10 #define trokis_dispatcher_func_hpp
11
12 #include <map>
13 #include <utility>
14 #include <functional>
15
16 namespace trokis { namespace utilities {
17
18     // Dispatcher functor. Strips out second argument of a function
           call and uses it
19     // as the index into a loopup table/map to determine the actual
           function to call.
20     template<typename>
21     class dispatcher;
22
23     template<typename SwitchType, typename RetType, typename...
           ArgTypes>
24     class dispatcher<RetType(packet_ptr&&, SwitchType, ArgTypes...)
           > {
25     public:
26         dispatcher() { }
27         dispatcher(const dispatcher&) = delete;
28
29         using result_type = RetType;
30         using func_type = RetType(packet_ptr&&, ArgTypes...);
31         using fallback_func_type = RetType(packet_ptr&&, SwitchType
               , ArgTypes...);
32         using wrapped_func_type = std::function<func_type>;
33         using fallback_wrapped_func_type = std::function<
               fallback_func_type>;
```

```
34          using map_type = std::map<SwitchType, wrapped_func_type>;
35
36          result_type operator()(packet_ptr&& p, SwitchType key,
                ArgTypes&&... args) {
37              auto func = find_handler(key);
38              if(func != funcs_.end()) {
39                  return std::get<1>(*func)(std::move(p), std::
                        forward<ArgTypes>(args)...);
40              } else {
41                  return fallback_(std::move(p), key, std::forward<
                        ArgTypes>(args)...);
42              }
43          }
44
45          template<typename Func>
46          void set_handler(SwitchType key, Func&& func) {
47              funcs_[key] = std::forward<Func>(func);
48          }
49
50          template<typename Func>
51          void set_fallback(Func&& func) {
52              fallback_ = std::forward<Func>(func);
53          }
54
55      private:
56          typename map_type::iterator find_handler(SwitchType key) {
57              return funcs_.find(key);
58          }
59
60          map_type funcs_;
61          fallback_wrapped_func_type fallback_;
62      };
63
64      // Default value for a hook with the given return type.
65      template<typename>
66      struct default_hook_return;
67
68      template<>
69      struct default_hook_return<packet_verdict> {
70          static constexpr auto value = packet_verdict::dropped;
71      };
72
73      // Default hook implementation, as a fallback when no key
            matched in a dispatcher.
74      template<typename RetType, typename... Args>
75      struct default_hook_impl;
76
77      template<typename RetType, typename... Args>
78      struct default_hook_impl<RetType(Args...)> {
79          RetType operator()(Args...) {
80              return default_hook_return<RetType>::value;
81          }
82      };
83
84      // Returns the default hook for a given function prototype.
```

```
85      template<typename Hook>
86      std::function<Hook> default_hook() {
87          return default_hook_impl<Hook>();
88      }
89
90      // Mixin for a generic request hook.
91      template<typename Request>
92      class generic_block_request_hook {
93      private:
94          using hook_request_t = std::function<Request>;
95      public:
96          generic_block_request_hook() :
97          hook_request_(default_hook<Request>())
98          { }
99
100         template<typename Func>
101         void set_hook_request(Func&& hook) {
102             hook_request_ = std::forward<Func>(hook);
103         }
104
105     protected:
106
107         template<typename... Args>
108         typename hook_request_t::result_type request(Args&&... args
                ) {
109             return hook_request_(std::forward<Args>(args)...);
110         }
111
112     private:
113         hook_request_t hook_request_;
114     };
115
116     // Mixin for a generic inform hook.
117     template<typename Inform>
118     class generic_block_inform_hook {
119     private:
120         using hook_inform_t = std::function<Inform>;
121     public:
122         generic_block_inform_hook() :
123         hook_inform_(default_hook<Inform>())
124         { }
125
126         template<typename Func>
127         void set_hook_inform(Func&& hook) {
128             hook_inform_ = std::forward<Func>(hook);
129         }
130
131     protected:
132
133         template<typename... Args>
134         typename hook_inform_t::result_type inform(Args&&... args)
                {
135             return hook_inform_(std::forward<Args>(args)...);
136         }
137
```

```
138     private:
139         hook_inform_t hook_inform_;
140     };
141 
142     // Mixin combining an Inform hook and a Request hook as a
            single mixin.
143     template<
144         typename Inform,
145         typename Request
146     >
147     class generic_block : public generic_block_inform_hook<Inform>,
148                           public generic_block_request_hook<Request
                                  >
149     { };
150 
151 } }
152 
153 #endif
```

Listing B.30: trokis/support/gc.hpp

```
1  //
2  //  gc.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 11/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8  
9  #ifndef trokis_gc_hpp
10 #define trokis_gc_hpp
11 
12 #include <memory>
13 #include <atomic>
14 #include <chrono>
15 #include <thread>
16 
17 #include "trokis/containers/intrusive_queue_mpsc.hpp"
18 #include "trokis/support/gc_reclaimer.hpp"
19 
20 namespace trokis { namespace gc {
21 
22     template<typename T>
23     class gc_ptr
24     {
25     public:
26         using element_type = T;
27 
28         // Constructs the pointer with a null value
29         gc_ptr() : ptr_(nullptr) { }
30 
31         // Constructs the pointer with a given value
32         gc_ptr(element_type* ptr) : ptr_(ptr) { }
33 
34         ~gc_ptr() {
```

```
35              if(ptr_ != nullptr) {
36                  reclaim(ptr_, static_default_delete<element_type>::
                        call);
37              }
38          }
39
40          // Move constructor
41          gc_ptr(gc_ptr<element_type>&& other) : ptr_(nullptr) {
42              assign(other.ptr_.load());
43              other.ptr_ = nullptr;
44          }
45
46          // Move assignment.
47          gc_ptr<element_type>& operator=(gc_ptr<element_type>&&
                other) {
48              assign(other.ptr_.load());
49              other.ptr_ = nullptr;
50              return *this;
51          }
52
53          // Acquires the pointer by performing a load with the
                proper memory barriers.
54          element_type* acquire() const {
55              return ptr_.load();
56          }
57
58          // Copies the pointed to object and performs the update in
                the supplied function object.
59          // Then tries to perform a CAS on the pointer value, and if
                 successful, returns. If the
60          // object was updated by another updater meanwhile, the
                function is restarted. In effect,
61          // a very simple form of software transactional memory is
                implemented.
62          //
63          // The supplied function object should NOT cause side
                effects, as it may be run multiple
64          // times.
65          //
66          // It is up to clients of this function to perform any
                locking that may be required to
67          // synchronize writers.
68          template<typename Func>
69          void update(Func&& f) {
70              while(1) {
71                  auto current = acquire();
72                  std::unique_ptr<element_type> updated;
73                  if(current) {
74                      updated = std::make_unique<element_type>(*
                            current);
75                  } else {
76                      updated = std::make_unique<element_type>();
77                  }
78                  f(*updated);
79                  if(compare_assign(current, updated.get())) {
```

```
80                  updated.release();
81                  return;
82              }
83          }
84      }
85
86      // Performs an atomic CAS of the pointer, with the default
            deleter for the old object.
87      bool compare_assign(element_type* expected, element_type*
            ptr) {
88          return compare_assign(expected, ptr,
                static_default_delete<element_type>::call);
89      }
90
91      // Performs an atomic CAS of the pointer, using a given
            deleter for the old object.
92      template<typename Deleter>
93      bool compare_assign(element_type* expected, element_type*
            ptr, Deleter&& deleter) {
94          auto success = ptr_.compare_exchange_weak(expected, ptr
                );
95          if(success) {
96              reclaim(expected, std::forward<Deleter>(deleter));
97              return true;
98          } else {
99              return false;
100         }
101     }
102
103     // Performs an atomic assignment of the pointer, using the
            default delter for the old object.
104     void assign(element_type* ptr) {
105         assign(ptr, static_default_delete<element_type>::call);
106     }
107
108     // Performs an atomic assignment of the pointer, using the
            given deleter for the old object.
109     template<typename Deleter>
110     void assign(element_type* ptr, Deleter&& deleter) {
111         auto oldptr = ptr_.exchange(ptr);
112         if(oldptr == nullptr) {
113             return;
114         }
115         reclaim(oldptr, std::forward<Deleter>(deleter));
116     }
117
118     // See assign(element_type*)
119     gc_ptr<element_type>& operator=(element_type* ptr) {
120         assign(ptr);
121         return *this;
122     }
123
124     // See acquire()
125     element_type& operator*() const {
126         return *acquire();
```

```
127         }
128
129         // See acquire()
130         element_type* operator->() const {
131             return acquire();
132         }
133
134     private:
135         template<typename Deleter>
136         void reclaim(element_type* ptr, Deleter&& deleter) {
137             __global_reclaimer.reclaim(ptr, std::forward<Deleter>(
                    deleter));
138         }
139
140         std::atomic<element_type*> ptr_;
141     };
142
143     // std::make_shared-like constructor for gc_ptr. Forwards all
            arguments directly to
144     // the constructor of the object type.
145     template<typename T, typename... Args>
146     gc_ptr<T> make_gc_ptr(Args&&... args) {
147         return gc_ptr<T>(new T(std::forward<Args>(args)...));
148     }
149
150 } }
151
152 #endif
```

Listing B.31: trokis/support/gc_reclaimer.hpp

```
1  //
2  //  gc_reclaimer.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 15/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_gc_reclaimer_hpp
10 #define trokis_gc_reclaimer_hpp
11
12 namespace trokis { namespace gc {
13
14     // Garbage collection reclamation object. Allows gc_ptrs to
           enqueue their objects and
15     // frees these after a set timeout(default 10 seconds) has
           passed. A future addition would
16     // be implementing a proxy collection scheme, allowing garbage
           to be held only until it is
17     // no longer needed, rather than for a given timeout. This
           would result in memory being
18     // reclaimed more quickly, but would also add more overhead to
           the code(as some sort of
```

```
19      // reference count would have to be maintained). The timeout
            based solution avoids this
20      // overhead at the expense of reclamation delay, and also
            ensures that memory can never be
21      // accidentally held forever.
22      class gc_reclaimer
23      {
24      private:
25          using reclaim_clock = std::chrono::steady_clock;
26
27          struct reclaimer_t {
28              reclaimer_t() : ptr(nullptr), inserted_at(reclaim_clock
                    ::now()), next(nullptr) { }
29              ~reclaimer_t() {
30                  if(ptr) {
31                      deleter(ptr); // Performs actual garbage
                            reclamation.
32                  }
33              }
34
35              void* ptr;
36              std::function<void(void*)> deleter;
37              reclaim_clock::time_point inserted_at;
38              reclaimer_t* next;
39          };
40
41          void gc_worker() {
42              while(!terminating_.load()) {
43                  auto current_item = reclaim_queue_.pop();
44                  if(!current_item) {
45                      // Sleep for one second
46                      std::this_thread::sleep_for(std::chrono::
                            seconds(1));
47                      continue;
48                  }
49                  auto threshold = reclaim_clock::now() - timeout_;
50                  while(current_item->inserted_at > threshold &&
                        !terminating_.load()) {
51                      std::this_thread::sleep_for(std::chrono::
                            seconds(1));
52                      threshold = reclaim_clock::now() - timeout_;
53                  }
54                  // RAII will delete for us, we simply pull out
                        items and wait until they can be deleted
55              }
56          }
57
58      public:
59          template<typename TimeUnit = std::chrono::seconds>
60          gc_reclaimer(TimeUnit object_timeout = std::chrono::seconds
                (10)) : timeout_(object_timeout) {
61              gc_thread_ = std::thread(&gc_reclaimer::gc_worker, this
                    );
62          }
63
```

```
64          ~gc_reclaimer() {
65              if(gc_thread_.joinable()) {
66                  terminating_ = true;
67                  gc_thread_.join();
68              }
69          }
70
71          template<typename T, typename Deleter>
72          void reclaim(T* ptr, Deleter deleter) {
73              auto reclaimer = std::make_unique<reclaimer_t>();
74              reclaimer->ptr = ptr;
75              reclaimer->deleter = std::forward<Deleter>(deleter);
76              reclaim_queue_.push(std::move(reclaimer));
77          }
78
79      private:
80          std::atomic<bool> terminating_;
81          std::thread gc_thread_;
82          containers::queue_mpsc<reclaimer_t> reclaim_queue_;
83          std::chrono::milliseconds timeout_;
84      };
85      static gc_reclaimer ___global_reclaimer;
86
87      template<typename T>
88      struct static_default_delete
89      {
90          static void call(void* arg) {
91              T* ptr = reinterpret_cast<T*>(arg);
92              std::default_delete<T>()(ptr);
93          }
94      };
95
96  } }
97
98  #endif
```

Listing B.32: trokis/support/neighbor_table.hpp

```
1   //
2   //  neighbor_table.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 13/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_neighbor_table_hpp
10  #define trokis_neighbor_table_hpp
11
12  #include "trokis/types.hpp"
13  #include "trokis/support/gc.hpp"
14
15  #include <map>
16  #include <cassert>
17  #include <memory>
```

```
18 #include <deque>
19 #include <chrono>
20 #include <mutex>
21
22 #include "trokis/protocols/ipv4/types.hpp"
23 #include "trokis/protocols/ieee802/types.hpp"
24
25 namespace trokis { namespace neighbor {
26
27     // Generic neighbor state machine. Encapsulates the state
          machine used for each neighbor
28     // in a neighbor table, and calls out to the given provider
          every time a packet(or solicitation)
29     // needs to be sent out.
30     template<typename Provider>
31     class generic_neighbor {
32     public:
33         using upper_address_type = typename Provider::
              upper_address_type;
34         using lower_address_type = typename Provider::
              lower_address_type;
35
36         static constexpr std::size_t QUEUE_LIMIT = 1000; // Maximum
               1000 packets queued for transmission.
37
38         generic_neighbor(Provider& provider, device_id devid, const
               upper_address_type& upper_addr)
39         : provider_(provider)
40         , devid_(devid)
41         , upper_address_(upper_addr)
42         , lower_valid_(false)
43         , sent_solicit_(false)
44         { }
45
46         // Tries to deliver a packet to the neighbor. If the lower
              address of the neighbor is
47         // not yet known, we instead enqueue the packet and ask the
               provider to solicit a request.
48         packet_verdict deliver(packet_ptr&& pkt) {
49             auto lower_addr = get_lower();
50             if(std::get<0>(lower_addr)) {
51                 // Output the packet
52                 return provider_.output(std::move(pkt), std::get
                       <2>(lower_addr), std::get<1>(lower_addr));
53             } else {
54                 // Queue the packet and ask for a solicit.
55                 query();
56                 enqueue_packet(std::move(pkt));
57                 return packet_verdict::success;
58             }
59         }
60
61         // Set the lower level addreses of the neighbor. The source
               address must be set too,
```

```
62          // but lower layers(or the provider itself) may of course
                ignore it if they choose.
63          void set_lower_address(const lower_address_type& addr,
                const lower_address_type& src) {
64              std::unique_lock<decltype(lower_addr_lock_)> guard(
                    lower_addr_lock_);
65              lower_valid_ = true;
66              lower_address_ = addr;
67              lower_address_src_ = src;
68              guard.unlock();
69
70              // Empty the queue of packets after dropping the lock
71              empty_queue(addr, src);
72          }
73
74      private:
75          using solicit_clock = std::chrono::steady_clock;
76
77          // Ask the Provider to solicit a request for our upper
                layer address. If a solicit was
78          // sent less than a second ago, we don't send one yet.
79          void query() {
80              using namespace std::chrono;
81              std::unique_lock<decltype(solicit_lock_)> guard(
                    solicit_lock_);
82              auto curtime = solicit_clock::now();
83              if(sent_solicit_) {
84                  if(duration_cast<milliseconds>(curtime -
                        last_solicit_time_).count() < 1000) {
85                      return;
86                  }
87              }
88              sent_solicit_ = true;
89              last_solicit_time_ = curtime;
90              // Drop the solicit lock here so other threads can
                    progress earlier
91              guard.unlock();
92              provider_.solicit(devid_, upper_address_);
93          }
94
95          // Enqueues a packet to be transmitted once the ARP solicit
                 is answered
96          void enqueue_packet(packet_ptr&& pkt) {
97              std::lock_guard<decltype(queue_lock_)> guard(
                    queue_lock_);
98              queue_.emplace_back(solicit_clock::now(), std::move(pkt
                    ));
99              while(queue_.size() > QUEUE_LIMIT) { // Limit the
                    maximum size of the queue
100                 queue_.pop_front();
101             }
102         }
103
104         // Empties the packet queue for this neighbor entry
```

```
105         void empty_queue(const lower_address_type& addr, const
                lower_address_type& src) {
106             using namespace std::chrono;
107             std::lock_guard<decltype(queue_lock_)> guard(
                    queue_lock_);
108             auto curtime = solicit_clock::now();
109             while(queue_.size() > 0) {
110                 // Send out the packets in the queue. If they were
                        enqueued more than
111                 // 500 ms ago, drop them.
112                 auto& front = queue_.front();
113                 if(duration_cast<milliseconds>(curtime - std::get
                        <0>(front)).count() < 500) {
114                     provider_.output(std::move(std::get<1>(front)),
                            src, addr);
115                 }
116                 queue_.pop_front();
117             }
118         }
119
120         // Retrieves the lower level address along with its
                validity field.
121         // OPTIMIZATION: Make this use CAS so the lower address can
                be retrieved without a lock,
122         // making the fast-path lock-less. For now, since the lock
                is extremely short, we will
123         // use this simpler solution.
124         std::tuple<bool,lower_address_type,lower_address_type>
                get_lower() {
125             std::lock_guard<decltype(lower_addr_lock_)> guard(
                    lower_addr_lock_);
126             return std::make_tuple(lower_valid_, lower_address_,
                    lower_address_src_);
127         }
128
129         Provider& provider_;
130         device_id devid_;
131         const upper_address_type upper_address_;
132
133         std::mutex lower_addr_lock_;
134         lower_address_type lower_address_;
135         lower_address_type lower_address_src_;
136         bool lower_valid_;
137
138         std::mutex queue_lock_;
139         // OPTIMIZATION: Change this to a ring buffer rather than a
                deque
140         std::deque<std::pair<solicit_clock::time_point, packet_ptr
                >> queue_;
141
142         std::mutex solicit_lock_;
143         solicit_clock::time_point last_solicit_time_;
144         bool sent_solicit_;
145     };
146
```

```
147     // Provides a neighborhood table backed by the given Provider.
148     //
149     // The provider specifies the upper and lower level addresses
            that it maps
150     // between, and performs this mapping for any device that is
            used with it.
151     template<typename Provider>
152     class neighbor_table {
153     public:
154         using provider_type = Provider;
155         using neighbor_type = generic_neighbor<Provider>;
156         using upper_address_type = typename provider_type::
                upper_address_type;
157         using lower_address_type = typename provider_type::
                lower_address_type;
158 
159         template<typename... Args>
160         neighbor_table(Args&&... args) : provider_(std::forward<
                Args>(args)...)
161         { }
162 
163         neighbor_type* get_or_create(const device_id devid, const
                upper_address_type& addr) {
164             std::lock_guard<decltype(neighbors_lock_)> guard(
                    neighbors_lock_);
165             auto key = std::make_tuple(devid, addr);
166             auto& item = neighbors_[key];
167             if(item.acquire() == nullptr) { // We are under lock,
                    don't need to do CAS
168                 item.assign(std::make_unique<neighbor_type>(
                        provider_, devid, addr).release());
169             }
170             return item.acquire();
171         }
172 
173         neighbor_type* get(const device_id devid, const
                upper_address_type& addr) {
174             std::lock_guard<decltype(neighbors_lock_)> guard(
                    neighbors_lock_);
175             auto key = std::make_tuple(devid, addr);
176             auto retval = neighbors_.find(key);
177             if(retval != neighbors_.end()) {
178                 return retval->second.acquire();
179             } else {
180                 return nullptr;
181             }
182         }
183 
184         void flush() {
185             std::lock_guard<decltype(neighbors_lock_)> guard(
                    neighbors_lock_);
186             neighbors_.clear();
187         }
188 
```

```
189         packet_verdict transmit(packet_ptr&& pkt, const
                upper_address_type& addr) {
190             auto neigh = get_or_create(pkt->dev, addr);
191             if(neigh == nullptr) {
192                 return packet_verdict::dropped;
193             }
194             return neigh->deliver(std::move(pkt));
195         }
196 
197         Provider& get_provider() {
198             return provider_;
199         }
200 
201     private:
202         Provider provider_;
203         std::map<std::tuple<device_id, upper_address_type>, gc::
                gc_ptr<neighbor_type>> neighbors_;
204         std::mutex neighbors_lock_;
205     };
206 
207 } }
208 
209 #endif
```

Listing B.33: trokis/support/pcap_output.hpp

```
1  //
2  //  pcap_dumper.h
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 14/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8 
9  #ifndef trokis_pcap_dumper_h
10 #define trokis_pcap_dumper_h
11 
12 #include <fstream>
13 #include <algorithm>
14 #include <array>
15 #include <thread>
16 #include <iterator>
17 #include <mutex>
18 
19 #include "trokis/packet.hpp"
20 #include "trokis/support/byte_tools.hpp"
21 
22 namespace trokis { namespace tools {
23 
24     enum pcap_encap {
25         ethernet = 1,
26         ipv4     = 228,
27         ipv6     = 229
28     };
29 
```

```
30      namespace detail {
31
32          struct pcap_output_wrapper {
33              pcap_output_wrapper(std::string filename, pcap_encap
                    encap_type)
34              : os(filename, std::ios::trunc | std::ios::binary | std
                    ::ios::out), osi(os), encap_(encap_type)
35              {
36                  using namespace trokis::byte_tools;
37                  write_bytes<endianness_host>((uint32_t)0xa1b2c3d4,
                        osi);
38                  write_bytes<endianness_host>((uint16_t)2, osi);
39                  write_bytes<endianness_host>((uint16_t)4, osi);
40                  write_bytes<endianness_host>((int32_t)0, osi);
41                  write_bytes<endianness_host>((uint32_t)0, osi);
42                  write_bytes<endianness_host>((uint32_t)~0, osi);
43                  write_bytes<endianness_host>((uint32_t)encap_, osi)
                        ;
44              }
45
46              template<typename Iterator>
47              void output(const Iterator& begin, const Iterator& end)
                     {
48                  using namespace trokis::byte_tools;
49
50                  std::lock_guard<std::mutex> lock(glob_lock);
51                  auto timestamp = std::chrono::microseconds(std::
                        chrono::seconds(std::time(NULL))).count();
52                  uint32_t length = uint32_t(std::distance(begin, end
                        ));
53
54                  write_bytes<endianness_host>((uint32_t)(timestamp /
                         1000000), osi);
55                  write_bytes<endianness_host>((uint32_t)(timestamp %
                         1000000), osi);
56                  write_bytes<endianness_host>(length, osi);
57                  write_bytes<endianness_host>(length, osi);
58
59                  std::copy(begin, end, osi);
60              }
61
62          private:
63
64              std::ofstream os;
65              std::ostream_iterator<uint8_t> osi;
66              std::mutex glob_lock;
67              pcap_encap encap_;
68          };
69
70      }
71
72      // Helper class for outputting packets to a .pcap file.
73      class pcap_output
74      {
75      public:
```

```
76          pcap_output(std::string filename, pcap_encap encap_type =
                pcap_encap::ethernet)
77          : wrapper_(std::make_shared<detail::pcap_output_wrapper>(
                filename, encap_type)) { }
78
79          pcap_output(const pcap_output& other) : wrapper_(other.
                wrapper_) { }
80
81          pcap_output(pcap_output&& other) : wrapper_(std::move(other
                .wrapper_)) { }
82
83          template<typename... Args>
84          packet_verdict operator()(packet_ptr&& pkt, Args&&... args)
                {
85              if(!pkt) {
86                  return packet_verdict::dropped;
87              }
88              auto& pdata = pkt->data();
89              wrapper_->output(pdata.begin(), pdata.end());
90              return packet_verdict::success;
91          }
92
93      private:
94          std::shared_ptr<detail::pcap_output_wrapper> wrapper_;
95      };
96
97  } }
98
99  #endif
```

Listing B.34: trokis/support/routing_table.hpp

```
1   //
2   //  routing_table.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 12/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_routing_table_hpp
10  #define trokis_routing_table_hpp
11
12  #include <algorithm>
13  #include <vector>
14
15  namespace trokis { namespace routing {
16
17      // Applies the given mask to the address. The address must be
            of a byte array-compatible type.
18      template<typename AddressType>
19      AddressType masked_address(AddressType addr, uint8_t netmask) {
20          uint8_t remaining = netmask;
21          auto cur = std::begin(addr);
22          constexpr auto part_len = sizeof(*cur);
```

```
23          for(auto& cur : addr) {
24              if(remaining == 0) {
25                  cur = 0;
26              } else if(remaining >= 8 * part_len) {
27                  remaining -= 8 * part_len;
28              } else {
29                  // Blank the last `remaining` bits of this section
30                  cur &= ~((1 << (8 * part_len - remaining)) - 1);
31                  remaining = 0;
32              }
33          }
34          return addr;
35      }
36
37      // Performs a longet-prefix-match between a route and and
            address.
38      template<typename AddressType>
39      bool route_match_lpm(AddressType addr, AddressType route,
            uint8_t netmask) {
40          return masked_address(addr, netmask) == route;
41      }
42
43      // Encapsulates a route.
44      template<typename AddressType>
45      struct route {
46          AddressType addr;
47          uint8_t mask;
48          bool direct;
49          AddressType nexthop_addr;
50          device_id dev;
51      };
52
53      // Simple routing table for trokis. Currently simply uses a
            list of routes and sequentially
54      // matches these. This will of course not scale to a large
            amount of routes, but is very simple
55      // and allows great performance for a low amount of routes. The
             routing table itself implements
56      // no locking, meaning that it is up to the client to ensure
            that access to the routing table is
57      // concurrency safe. The recommended approach(and the one used
            in the trokis clients, IPv4 and IPv6)
58      // is to make a complete copy of the routing table, update the
            copy and finally atomically commit the
59      // table using a compare-and-swap. This allows wait-free
            waiting for readers and lock-free access for
60      // writers.
61      template<typename AddressType>
62      class routing_table {
63      public:
64          using route_type = route<AddressType>;
65
66          void add_route(const route_type& route) {
67              routes_.push_back(route);
68              auto& back = routes_.back();
```

```
69              back.addr = masked_address(back.addr, back.mask);
70          }
71
72          void del_route(const route_type& route) {
73              std::remove(std::begin(routes_), std::end(routes_),
                    route);
74          }
75
76          const route_type* find_route_lpm(const AddressType addr)
                const {
77              const route_type* curbest = nullptr;
78              for(auto& route : routes_) {
79                  if(route_match_lpm(addr, route.addr, route.mask)) {
80                      if(curbest != nullptr && curbest->mask > route.
                            mask) {
81                          continue;
82                      }
83                      curbest = &route;
84                  }
85              }
86              return curbest;
87          }
88
89      private:
90          std::vector<route_type> routes_;
91      };
92
93  } }
94
95  #endif
```

Listing B.35: trokis/types.hpp

```
1   //
2   //  types.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 09/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_types_hpp
10  #define trokis_types_hpp
11
12  #include "trokis/detail/compat.h"
13
14  #include <memory>
15
16  namespace trokis {
17
18      using byte_t = uint8_t;
19
20      // Forward declarations
21      class packet;
22      class netdevice;
```

```
23      class worker;
24
25      // Generic packet verdicts
26      enum class packet_verdict {
27          success,         // Packet was received or buffered
28          dropped          // Packet was(should be) dropped
29      };
30
31      // Generic address type information
32      enum class address_type {
33          unicast,
34          multicast,
35          broadcast
36      };
37
38      // Pointer to a packet
39      using packet_ptr = std::unique_ptr<packet>;
40
41      // ID types
42      using worker_id = int;
43      using device_id = int;
44  }
45
46  #include "trokis/support/gc.hpp"
47
48  #endif
```

Appendix C

Trokis test implementation with extra optimizations

Listing C.1: bench/bench.cpp

```
#include <iostream>
#include <cstdint>
#include <vector>
#include <array>
#include <algorithm>
#include <functional>
#include <chrono>
#include <unordered_map>
#include <map>
#include <tuple>
#include <atomic>
#include <fstream>

#include <iomanip>
#include <sstream>
#include <type_traits>
#include <iterator>

template<std::size_t N>
std::ostream& operator<<(std::ostream& os, const std::array<uint8_t, N>& array) {
    bool first = true;
    for(auto p : array) {
        if(!first) {
            os << ".";
```

```
25          }
26          os << (int)p;
27          first = false;
28      }
29      return os;
30  }
31
32
33  #include "trokis/types.hpp"
34  #include "trokis/packet_data.hpp"
35  #include "trokis/support/byte_tools.hpp"
36  #include "trokis/packet.hpp"
37  #include "trokis/support/dispatcher.hpp"
38  #include "trokis/protocols/ipv4/ipv4.hpp"
39  #include "trokis/protocols/ieee802/ieee802_3.hpp"
40  #include "trokis/protocols/ipv4/arp.hpp"
41
42  #include "trokis/support/pcap_output.hpp"
43
44  #include <pcap.h>
45
46  using namespace trokis;
47
48  std::atomic<bool> running { true };
49
50  bool load_packets(std::string filename, std::vector<packet_ptr>&
        packets) {
51      char errbuf[PCAP_ERRBUF_SIZE];
52      auto pcap = pcap_open_offline(filename.c_str(), &errbuf[0]);
53      if(pcap == nullptr) {
54          std::cerr << "Failed to open pcap dump: " << errbuf << std
                ::endl;
55          return false;
56      }
57
58      while(1) {
59          struct pcap_pkthdr hdr;
60          auto pdata = pcap_next(pcap, &hdr);
61          if(pdata == nullptr) break;
62          auto pkt = std::make_unique<packet>();
63          auto pkt_data = std::make_unique<packet_data>(pdata, pdata
                + hdr.caplen);
64          pkt->replace_data(std::move(pkt_data));
65          packets.push_back(std::move(pkt));
66      }
67      return true;
68  }
69
70  __attribute__((noinline)) void run_benchmark(std::vector<packet_ptr
        >& packets, ieee802::ethernet_802_3& layer_mac, device_id devid
        ) {
71      std::size_t totcount = 0;
72      std::size_t pktcount = 0;
73      auto start = std::chrono::high_resolution_clock::now();
74      for(auto& pkt : packets) {
```

```
75          pkt->dev = devid;
76          pkt->worker = 1;
77          if(layer_mac.receive(std::move(pkt)) == packet_verdict::
                success) {
78              ++pktcount;
79          }
80          ++totcount;
81      }
82      auto end = std::chrono::high_resolution_clock::now();
83      auto diff = std::chrono::duration_cast<std::chrono::
            milliseconds>(end - start);
84      std::cerr << "Injected " << pktcount << "/" << totcount << "
            packets in " << diff.count() << " ms" << std::endl;
85      std::cout << diff.count() << std::endl;
86  }
87
88  int main(int argc, char *argv[]) {
89      /*** Initialize network stack ***/
90      using namespace std::placeholders;
91
92      /* Set up MAC layer for ethernet */
93      ieee802::ethernet_802_3 layer_mac;
94
95      /* Set up receive path ethertype dispatch */
96      utilities::dispatcher<packet_verdict(packet_ptr&&, ieee802::
            ethertype, address_type)> dispatch;
97      dispatch.set_fallback([](packet_ptr&&, ieee802::ethertype,
            address_type) { return packet_verdict::dropped; });
98      layer_mac.set_hook_inform(std::ref(dispatch));
99      layer_mac.set_hook_request([](packet_ptr&&) { return
            packet_verdict::success; });
100
101     ipv4::ipv4_protocol layer_ipv4;
102     layer_ipv4.set_gateway(true);
103
104     utilities::dispatcher<packet_verdict(packet_ptr&&, ipv4::
            protonum_t, ipv4::address_t, ipv4::address_t)>
            ipv4_dispatch;
105     ipv4_dispatch.set_fallback([](packet_ptr&&, ipv4::protonum_t,
            ipv4::address_t, ipv4::address_t) {
106         return packet_verdict::dropped;
107     });
108
109     layer_ipv4.set_hook_inform(std::ref(ipv4_dispatch));
110     neighbor::neighbor_table<neighbor::arp_provider<ipv4::address_t
            ,ieee802::dev_info_802_3>> neigh_ipv4_ethernet;
111     neigh_ipv4_ethernet.get_provider().set_output_hook(std::bind(&
            decltype(layer_mac)::transmit, &layer_mac, _1, _2, _3));
112     neigh_ipv4_ethernet.get_provider().set_primary_address_hook
            ([&](device_id devid) {
113         return std::make_pair(layer_ipv4.get_primary_address(devid)
                , layer_mac.get_primary_address(devid));
114     });
115     // Remember to switch to a demultiplexer here, if we start
            using lower level protocols that don't use MAC addresses.
```

```
116     layer_ipv4.set_hook_request(std::bind(&decltype(
            neigh_ipv4_ethernet)::transmit, &neigh_ipv4_ethernet, _1,
            _2));
117
118     /* Set up ARP receiver */
119     arp::arp_receiver layer_arp;
120     layer_arp.set_hook_request([&] (packet_ptr&& pkt, ieee802::
            mac_address src, ieee802::mac_address dst) {
121         pkt->data().push(2);
122         pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
123         return layer_mac.transmit(std::move(pkt), src, dst);
124     });
125     layer_arp.set_primary_address_hook(std::bind(&decltype(
            layer_mac)::get_primary_address, &layer_mac, _1));
126     layer_arp.set_is_local_address_hook(std::bind(&decltype(
            layer_ipv4)::is_local_address, &layer_ipv4, _1, _2));
127     layer_arp.set_update_neighbor_hook([&](device_id devid, ipv4::
            address_t upper, ieee802::mac_address lower, bool
            may_create) {
128         decltype(neigh_ipv4_ethernet.get(devid, upper)) neighbor;
129         if(may_create) {
130             neighbor = neigh_ipv4_ethernet.get_or_create(devid,
                    upper);
131         } else {
132             neighbor = neigh_ipv4_ethernet.get(devid, upper);
133         }
134         if(neighbor == nullptr) return false;
135         neighbor->set_lower_address(lower, ieee802::mac_address());
136         return true;
137     });
138
139     /* Set up ICMPv4 */
140     ipv4::icmp4::icmp4_protocol layer_ipv4_icmp;
141     layer_ipv4_icmp.set_hook_request(std::bind(&decltype(layer_ipv4
            )::transmit, &layer_ipv4, _1, _2, _3, _4));
142     ipv4_dispatch.set_handler(1, std::bind(&decltype(
            layer_ipv4_icmp)::receive, &layer_ipv4_icmp, _1, _2, _3));
143
144     // Set up dispatcher
145     dispatch.set_handler(0x0800, std::bind(&ipv4::ipv4_protocol::
            receive, &layer_ipv4, _1, _2));
146     dispatch.set_handler(0x0806, std::bind(&arp::arp_receiver::
            receive, &layer_arp, _1, _2));
147
148     // Set up devices
149     device_id dev1 = 1;
150     device_id dev2 = 2;
151     {
152         auto dev = layer_mac.get_or_create_dev_data(dev1);
153         dev->unicast_address_add({{0x01,0x23,0x45,0x67,0x89,0xab}})
                ;
154
155         auto v4 = layer_ipv4.get_or_create_dev_data(dev1);
156         v4->local_address_add({{10, 0, 1, 10}});
157
```

```
158     }
159     {
160         auto dev = layer_mac.get_or_create_dev_data(dev2);
161         dev->unicast_address_add({{0x00,0x23,0x14,0x30,0xce,0x74}})
                ;
162 
163         auto v4 = layer_ipv4.get_or_create_dev_data(dev2);
164         v4->local_address_add({{10, 0, 2, 10}});
165 
166         auto neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,5}}));
167         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
168         neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,1}}));
169         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
170     }
171     layer_ipv4.get_routing_table().add_route({{{10,0,1,0}}, 24,
            true, {{0,0,0,0}}, dev1});
172     layer_ipv4.get_routing_table().add_route({{{10,0,2,0}}, 24,
            true, {{0,0,0,0}}, dev2});
173     layer_ipv4.get_routing_table().add_route({{{0,0,0,0}}, 0, false
            , {{10,0,2,1}}, dev2});
174 
175     // Load test data
176     std::vector<packet_ptr> packets;
177     if(!load_packets("test.pcap", packets)) {
178         return 1;
179     }
180     std::cerr << "Loaded " << packets.size() << " packets" << std::
            endl;
181 
182     run_benchmark(packets, layer_mac, dev1);
183 
184     return 0;
185 }
```

Listing C.2: bench/bench.cpp

```
1  #include <iostream>
2  #include <cstdint>
3  #include <vector>
4  #include <array>
5  #include <algorithm>
6  #include <functional>
7  #include <chrono>
8  #include <unordered_map>
9  #include <map>
10 #include <tuple>
11 #include <atomic>
12 #include <fstream>
13 
14 #include <iomanip>
15 #include <sstream>
```

```
16 #include <type_traits>
17 #include <iterator>
18
19 template<std::size_t N>
20 std::ostream& operator<<(std::ostream& os, const std::array<uint8_t
       , N>& array) {
21     bool first = true;
22     for(auto p : array) {
23         if(!first) {
24             os << ".";
25         }
26         os << (int)p;
27         first = false;
28     }
29     return os;
30 }
31
32
33 #include "trokis/types.hpp"
34 #include "trokis/packet_data.hpp"
35 #include "trokis/support/byte_tools.hpp"
36 #include "trokis/packet.hpp"
37 #include "trokis/support/dispatcher.hpp"
38 #include "trokis/protocols/ipv4/ipv4.hpp"
39 #include "trokis/protocols/ieee802/ieee802_3.hpp"
40 #include "trokis/protocols/ipv4/arp.hpp"
41
42 #include "trokis/support/pcap_output.hpp"
43
44 #include <pcap.h>
45
46 using namespace trokis;
47
48 std::atomic<bool> running { true };
49
50 bool load_packets(std::string filename, std::vector<packet_ptr>&
       packets) {
51     char errbuf[PCAP_ERRBUF_SIZE];
52     auto pcap = pcap_open_offline(filename.c_str(), &errbuf[0]);
53     if(pcap == nullptr) {
54         std::cerr << "Failed␣to␣open␣pcap␣dump:␣" << errbuf << std
               ::endl;
55         return false;
56     }
57
58     while(1) {
59         struct pcap_pkthdr hdr;
60         auto pdata = pcap_next(pcap, &hdr);
61         if(pdata == nullptr) break;
62         auto pkt = std::make_unique<packet>();
63         auto pkt_data = std::make_unique<packet_data>(pdata, pdata
               + hdr.caplen);
64         pkt->replace_data(std::move(pkt_data));
65         packets.push_back(std::move(pkt));
66     }
```

```
67      return true;
68  }
69
70  __attribute__((noinline)) void run_benchmark(std::vector<packet_ptr
        >& packets, ieee802::ethernet_802_3& layer_mac, device_id devid
        ) {
71      std::size_t totcount = 0;
72      std::size_t pktcount = 0;
73      auto start = std::chrono::high_resolution_clock::now();
74      for(auto& pkt : packets) {
75          pkt->dev = devid;
76          pkt->worker = 1;
77          if(layer_mac.receive(std::move(pkt)) == packet_verdict::
                success) {
78              ++pktcount;
79          }
80          ++totcount;
81      }
82      auto end = std::chrono::high_resolution_clock::now();
83      auto diff = std::chrono::duration_cast<std::chrono::
            milliseconds>(end - start);
84      std::cerr << "Injected " << pktcount << "/" << totcount << "
            packets in " << diff.count() << " ms" << std::endl;
85      std::cout << diff.count() << std::endl;
86  }
87
88  int main(int argc, char *argv[]) {
89      /*** Initialize network stack ***/
90      using namespace std::placeholders;
91
92      /* Set up MAC layer for ethernet */
93      ieee802::ethernet_802_3 layer_mac;
94
95      /* Set up receive path ethertype dispatch */
96      utilities::dispatcher<packet_verdict(packet_ptr&&, ieee802::
            ethertype, address_type)> dispatch;
97      dispatch.set_fallback([](packet_ptr&&, ieee802::ethertype,
            address_type) { return packet_verdict::dropped; });
98      layer_mac.set_hook_inform(std::ref(dispatch));
99      layer_mac.set_hook_request([](packet_ptr&&) { return
            packet_verdict::success; });
100
101     ipv4::ipv4_protocol layer_ipv4;
102     layer_ipv4.set_gateway(true);
103
104     utilities::dispatcher<packet_verdict(packet_ptr&&, ipv4::
            protonum_t, ipv4::address_t, ipv4::address_t)>
            ipv4_dispatch;
105     ipv4_dispatch.set_fallback([](packet_ptr&&, ipv4::protonum_t,
            ipv4::address_t, ipv4::address_t) {
106         return packet_verdict::dropped;
107     });
108
109     layer_ipv4.set_hook_inform(std::ref(ipv4_dispatch));
```

```
110     neighbor::neighbor_table<neighbor::arp_provider<ipv4::address_t
            ,ieee802::dev_info_802_3>> neigh_ipv4_ethernet;
111     neigh_ipv4_ethernet.get_provider().set_output_hook(std::bind(&
            decltype(layer_mac)::transmit, &layer_mac, _1, _2, _3));
112     neigh_ipv4_ethernet.get_provider().set_primary_address_hook
            ([&](device_id devid) {
113         return std::make_pair(layer_ipv4.get_primary_address(devid)
                , layer_mac.get_primary_address(devid));
114     });
115     // Remember to switch to a demultiplexer here, if we start
            using lower level protocols that don't use MAC addresses.
116     layer_ipv4.set_hook_request(std::bind(&decltype(
            neigh_ipv4_ethernet)::transmit, &neigh_ipv4_ethernet, _1,
            _2));
117
118     /* Set up ARP receiver */
119     arp::arp_receiver layer_arp;
120     layer_arp.set_hook_request([&] (packet_ptr&& pkt, ieee802::
            mac_address src, ieee802::mac_address dst) {
121         pkt->data().push(2);
122         pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
123         return layer_mac.transmit(std::move(pkt), src, dst);
124     });
125     layer_arp.set_primary_address_hook(std::bind(&decltype(
            layer_mac)::get_primary_address, &layer_mac, _1));
126     layer_arp.set_is_local_address_hook(std::bind(&decltype(
            layer_ipv4)::is_local_address, &layer_ipv4, _1, _2));
127     layer_arp.set_update_neighbor_hook([&](device_id devid, ipv4::
            address_t upper, ieee802::mac_address lower, bool
            may_create) {
128         decltype(neigh_ipv4_ethernet.get(devid, upper)) neighbor;
129         if(may_create) {
130             neighbor = neigh_ipv4_ethernet.get_or_create(devid,
                    upper);
131         } else {
132             neighbor = neigh_ipv4_ethernet.get(devid, upper);
133         }
134         if(neighbor == nullptr) return false;
135         neighbor->set_lower_address(lower, ieee802::mac_address());
136         return true;
137     });
138
139     /* Set up ICMPv4 */
140     ipv4::icmp4::icmp4_protocol layer_ipv4_icmp;
141     layer_ipv4_icmp.set_hook_request(std::bind(&decltype(layer_ipv4
            )::transmit, &layer_ipv4, _1, _2, _3, _4));
142     ipv4_dispatch.set_handler(1, std::bind(&decltype(
            layer_ipv4_icmp)::receive, &layer_ipv4_icmp, _1, _2, _3));
143
144     // Set up dispatcher
145     dispatch.set_handler(0x0800, std::bind(&ipv4::ipv4_protocol::
            receive, &layer_ipv4, _1, _2));
146     dispatch.set_handler(0x0806, std::bind(&arp::arp_receiver::
            receive, &layer_arp, _1, _2));
147
```

```
148     // Set up devices
149     device_id dev1 = 1;
150     device_id dev2 = 2;
151     {
152         auto dev = layer_mac.get_or_create_dev_data(dev1);
153         dev->unicast_address_add({{0x01,0x23,0x45,0x67,0x89,0xab}})
                ;
154 
155         auto v4 = layer_ipv4.get_or_create_dev_data(dev1);
156         v4->local_address_add({{10, 0, 1, 10}});
157 
158     }
159     {
160         auto dev = layer_mac.get_or_create_dev_data(dev2);
161         dev->unicast_address_add({{0x00,0x23,0x14,0x30,0xce,0x74}})
                ;
162 
163         auto v4 = layer_ipv4.get_or_create_dev_data(dev2);
164         v4->local_address_add({{10, 0, 2, 10}});
165 
166         auto neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,5}}));
167         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
168         neigh = neigh_ipv4_ethernet.get_or_create(dev2, ipv4::
                address_t({{10,0,2,1}}));
169         neigh->set_lower_address(ieee802::mac_address
                ({{1,2,3,1,2,3}}), {{0x00,0x23,0x14,0x30,0xce,0x74}});
170     }
171     layer_ipv4.get_routing_table().add_route({{{10,0,1,0}}, 24,
            true, {{0,0,0,0}}, dev1});
172     layer_ipv4.get_routing_table().add_route({{{10,0,2,0}}, 24,
            true, {{0,0,0,0}}, dev2});
173     layer_ipv4.get_routing_table().add_route({{{0,0,0,0}}, 0, false
            , {{10,0,2,1}}, dev2});
174 
175     // Load test data
176     std::vector<packet_ptr> packets;
177     if(!load_packets("test.pcap", packets)) {
178         return 1;
179     }
180     std::cerr << "Loaded " << packets.size() << " packets" << std::
            endl;
181 
182     run_benchmark(packets, layer_mac, dev1);
183 
184     return 0;
185 }
```

Listing C.3: bench/bench_loop

```
1 #!/bin/sh
2 
3 echo "" > results.txt
4 
```

```
for i in {1..100}
do
    ./bench >> results.txt
done
```

Listing C.4: bench/bootstrap_mean.r

```
# Constants
rounds = 100000 # Number of bootstrap rounds(Monte-carlo case resampling)

# Load numbers from results file. Each result should have a seperate line.
data <- scan('results.txt')

# Perform bootstrap resampling
means <- sapply(c(1:rounds), function(x) sample(data, length(data), replace=TRUE))

# Output mean and standard deviation
cat(sprintf("Mean: %f", mean(means)))
cat(sprintf("SD:   %f", sd(means)))

# Find confidence interval on the bootstrapped data
confi <- quantile(means, probs = c(0.025, 0.975));
cat(sprintf("95%% conf. interval: %f <= %f <= %f", confi[1], mean(means), confi[2]))
```

Listing C.5: bench/build_clang

```
#!/bin/sh
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
pushd $DIR
time clang++ --std=c++11 bench.cpp -I.. -lpcap -o bench -Wfatal-errors -g -pthread -O3 -lpthread -Wall -stdlib=libc++ -lc++abi
# time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-errors -g -pthread -O0 -lpthread -Wall -stdlib=libc++ -lc++abi
RETVAL=$?
popd
exit $RETVAL
```

Listing C.6: bench/build_gcc

```
#!/bin/sh
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
pushd $DIR
time g++ --std=c++11 bench.cpp -I.. -lpcap -o bench -Wfatal-errors -g -pthread -O3 -lpthread -Wall
# time g++ --std=c++11 bench.cpp -I.. -lpcap -o bench -Wfatal-errors -g -pthread -lpthread -Wall
popd
```

Listing C.7: bench/build_generator

```
#!/bin/sh
```

```
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
pushd $DIR
time clang++ --std=c++11 generator.cpp -I.. -lpcap -o generator -Wfatal-errors -g -pthread -O3 -lpthread -Wall -stdlib=libc++ -lc++abi
# time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-errors -g -pthread -O0 -lpthread -Wall -stdlib=libc++ -lc++abi
RETVAL=$?
popd
exit $RETVAL
```

Listing C.8: bench/generator.cpp

```
#include <iostream>
#include <fstream>
#include <random>
#include <array>
#include <iterator>
#include <algorithm>

#include "trokis/support/byte_tools.hpp"

using namespace trokis::byte_tools;

constexpr std::size_t packet_count = 20e6;
constexpr std::size_t packet_size = 64;
constexpr std::size_t zerofill = packet_size - 6 * 2 - 2 - 20;

int main(int argc, char *argv[])
{
    std::ranlux48_base prng;
    prng.seed(std::random_device()());
    std::uniform_int_distribution<> dist(0, 99);
    auto next_rand = [&] () {
        return dist(prng);
    };

    std::ofstream output("test.pcap", std::ios_base::trunc
        | std::ios_base::out);
    std::ostream_iterator<uint8_t> f(output);

    // Fill in pcap header
    write_bytes<endianness_host>((uint32_t)0xa1b2c3d4, f); // Magic number
    write_bytes<endianness_host>((uint16_t)2, f);          // Major version
    write_bytes<endianness_host>((uint16_t)4, f);          // Minor version
    write_bytes<endianness_host>((int32_t)0, f);           // Timestamp in UTC
    write_bytes<endianness_host>((uint32_t)0, f);          // Timestamp accurary
    write_bytes<endianness_host>((uint32_t)~0, f);         // Snapshot length
```

```
36      write_bytes<endianness_host>((uint32_t)1, f);           //
            Ethernet link-layer type
37
38      // Create the base packet that we will modify and inject
            multiple times
39      std::array<uint8_t, packet_size> packet;
40      std::fill(std::begin(packet), std::end(packet), 0);
41
42      // Fill in ethernet header
43      write_bytes_array(std::array<uint8_t,6>({{0x1a,0x2b,0x3c,0x4d,0
            x5e,0x6f}}),
44          packet.begin() + 6);
45      write_bytes<endianness_big>((uint16_t)0x0800, packet.begin() +
            12);
46
47      // Fill in IP header
48      write_bytes<endianness_big>((uint8_t)0x45, packet.begin() + 14)
            ;
49      write_bytes<endianness_big>((uint16_t)50, packet.begin() + 16);
50      write_bytes<endianness_big>((uint8_t)64, packet.begin() + 22);
51      write_bytes<endianness_big>((uint8_t)17, packet.begin() + 23);
52      write_bytes_array(std::array<uint8_t,4>({{10,0,1,5}}), packet.
            begin() + 26);
53
54      // Fill the rest of the packet with UDP data
55      write_bytes<endianness_big>((uint16_t)0x7a69, packet.begin() +
            36);
56      write_bytes<endianness_big>((uint16_t)30, packet.begin() + 38);
57
58      for(std::size_t i = 0; i < packet_count; ++i) {
59          // Write pcap packet header
60          write_bytes<endianness_host>((uint32_t)0, f);
61          write_bytes<endianness_host>((uint32_t)0, f);
62          write_bytes<endianness_host>((uint32_t)packet_size, f);
63          write_bytes<endianness_host>((uint32_t)packet_size, f);
64
65          // Determine which kind of packet this will be
66          auto pkt_type = next_rand();
67
68          // Fill in dest MAC address
69          write_bytes_array(std::array<uint8_t,6>({{0x01,0x23,0x45,0
                x67,0x89,0xab}}), packet.begin() + 0);
70
71          if(pkt_type >= 55) {
72              // 45% to global IP(via nexthop)
73              write_bytes_array(std::array<uint8_t,4>({{1,2,3,4}}),
                    packet.begin() + 30);
74          } else if(pkt_type >= 10) {
75              // 45% to directly connected IP
76              write_bytes_array(std::array<uint8_t,4>({{10,0,2,5}}),
                    packet.begin() + 30);
77          } else if(pkt_type >= 5) {
78              // 5% Local received
79              write_bytes_array(std::array<uint8_t,4>({{10,0,1,10}}),
                     packet.begin() + 30);
```

```
80          } else {
81              // 5% Dropped at MAC layer
82              write_bytes_array(std::array<uint8_t,4>({{10,0,1,123}})
                    , packet.begin() + 30);
83              write_bytes_array(std::array<uint8_t,6>({{0xba,0x98,0
                    x76,0x54,0x32,0x10}}), packet.begin() + 0);
84          }
85          // Recalc the IP checksum
86          write_bytes<endianness_big>((uint16_t)0, packet.begin() +
                24);
87          auto csum = checksum_ones_complement(packet.begin() + 14,
                packet.begin() + 34);
88          write_bytes<endianness_big>((uint16_t)csum, packet.begin()
                + 24);
89
90          // Write ethernet header
91          std::copy(std::begin(packet), std::end(packet), f);
92      }
93
94      output.close();
95
96      return 0;
97  }
```

Listing C.9: functional_test/build_clang

```
1  #!/bin/sh
2  DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
3  pushd $DIR
4  time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
       errors -g -pthread -O3 -lpthread -Wall -stdlib=libc++ -lc++abi
5  # time clang++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
       errors -g -pthread -O0 -lpthread -Wall -stdlib=libc++ -lc++abi
6  RETVAL=$?
7  popd
8  exit $RETVAL
```

Listing C.10: functional_test/build_gcc

```
1  #!/bin/sh
2  DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
3  pushd $DIR
4  time g++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-errors
       -g -pthread -O3 -lpthread -Wall
5  # time g++ --std=c++11 main.cpp -I.. -lpcap -o trokis -Wfatal-
       errors -g -pthread -lpthread -Wall
6  popd
```

Listing C.11: functional_test/main.cpp

```
1  #include <iostream>
2  #include <cstdint>
3  #include <vector>
4  #include <array>
5  #include <algorithm>
```

```
6  #include <functional>
7  #include <chrono>
8  #include <unordered_map>
9  #include <map>
10 #include <tuple>
11 #include <atomic>
12 #include <fstream>
13
14 #include <iomanip>
15 #include <sstream>
16 #include <type_traits>
17 #include <iterator>
18
19 template<std::size_t N>
20 std::ostream& operator<<(std::ostream& os, const std::array<uint8_t
       , N>& array) {
21     bool first = true;
22     for(auto p : array) {
23         if(!first) {
24             os << ".";
25         }
26         os << (int)p;
27         first = false;
28     }
29     return os;
30 }
31
32
33 #include "trokis/types.hpp"
34 #include "trokis/packet_data.hpp"
35 #include "trokis/support/byte_tools.hpp"
36 #include "trokis/packet.hpp"
37 #include "trokis/support/dispatcher.hpp"
38 #include "trokis/protocols/ipv4/ipv4.hpp"
39 #include "trokis/protocols/ipv6/ipv6.hpp"
40 #include "trokis/protocols/ieee802/ieee802_3.hpp"
41 #include "trokis/protocols/ipv4/arp.hpp"
42
43 #include "trokis/support/pcap_output.hpp"
44
45 #include "trokis/devices/raw/netdevice_raw.hpp"
46
47 #include <pcap.h>
48
49 #include <signal.h>
50
51 using namespace trokis;
52
53 std::atomic<bool> running { true };
54
55 int main(int argc, char *argv[]) {
56     /*** Initialize network stack ***/
57     using namespace std::placeholders;
58
59     /* Set up MAC layer for ethernet */
```

```
60      ieee802::ethernet_802_3 layer_mac;
61
62      /* Set up receive path ethertype dispatch */
63      utilities::dispatcher<packet_verdict(packet_ptr&&, ieee802::
            ethertype, address_type)> dispatch;
64      dispatch.set_fallback([](packet_ptr&&, ieee802::ethertype,
            address_type) { return packet_verdict::dropped; });
65      layer_mac.set_hook_inform(std::ref(dispatch));
66      // layer_mac.set_hook_request(tools::pcap_output("ethernet.pcap
            "));
67
68      ipv4::ipv4_protocol layer_ipv4;
69
70      utilities::dispatcher<packet_verdict(packet_ptr&&, ipv4::
            protonum_t, ipv4::address_t, ipv4::address_t)>
            ipv4_dispatch;
71      ipv4_dispatch.set_fallback([](packet_ptr&&, ipv4::protonum_t,
            ipv4::address_t, ipv4::address_t) {
72          return packet_verdict::dropped;
73      });
74
75      layer_ipv4.set_hook_inform(std::ref(ipv4_dispatch));
76      neighbor::neighbor_table<neighbor::arp_provider<ipv4::address_t
            ,ieee802::dev_info_802_3>> neigh_ipv4_ethernet;
77      neigh_ipv4_ethernet.get_provider().set_output_hook(std::bind(&
            decltype(layer_mac)::transmit, &layer_mac, _1, _2, _3));
78      neigh_ipv4_ethernet.get_provider().set_primary_address_hook
            ([&](device_id devid) {
79          return std::make_pair(layer_ipv4.get_primary_address(devid)
                , layer_mac.get_primary_address(devid));
80      });
81      // Remember to switch to a demultiplexer here, if we start
            using lower level protocols that don't use MAC addresses.
82      layer_ipv4.set_hook_request(std::bind(&decltype(
            neigh_ipv4_ethernet)::transmit, &neigh_ipv4_ethernet, _1,
            _2));
83
84      /* Set up ARP receiver */
85      arp::arp_receiver layer_arp;
86      layer_arp.set_hook_request([&] (packet_ptr&& pkt, ieee802::
            mac_address src, ieee802::mac_address dst) {
87          pkt->data().push(2);
88          pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
89          return layer_mac.transmit(std::move(pkt), src, dst);
90      });
91      layer_arp.set_primary_address_hook(std::bind(&decltype(
            layer_mac)::get_primary_address, &layer_mac, _1));
92      layer_arp.set_is_local_address_hook(std::bind(&decltype(
            layer_ipv4)::is_local_address, &layer_ipv4, _1, _2));
93      layer_arp.set_update_neighbor_hook([&](device_id devid, ipv4::
            address_t upper, ieee802::mac_address lower, bool
            may_create) {
94          decltype(neigh_ipv4_ethernet.get(devid, upper)) neighbor;
95          if(may_create) {
```

```
96              neighbor = neigh_ipv4_ethernet.get_or_create(devid,
                    upper);
97          } else {
98              neighbor = neigh_ipv4_ethernet.get(devid, upper);
99          }
100         if(neighbor == nullptr) return false;
101         neighbor->set_lower_address(lower, ieee802::mac_address());
102         return true;
103     });
104
105     /* Set up ICMPv4 */
106     ipv4::icmp4::icmp4_protocol layer_ipv4_icmp;
107     layer_ipv4_icmp.set_hook_request(std::bind(&decltype(layer_ipv4
            )::transmit, &layer_ipv4, _1, _2, _3, _4));
108     ipv4_dispatch.set_handler(1, std::bind(&decltype(
            layer_ipv4_icmp)::receive, &layer_ipv4_icmp, _1, _2, _3));
109
110     /* Set up IPv6 */
111     ipv6::ipv6_protocol layer_ipv6;
112
113     // Set up dispatcher
114     dispatch.set_handler(0x0800, std::bind(&ipv4::ipv4_protocol::
            receive, &layer_ipv4, _1, _2));
115     dispatch.set_handler(0x86DD, std::bind(&ipv6::ipv6_protocol::
            receive, &layer_ipv6, _1, _2));
116     dispatch.set_handler(0x0806, std::bind(&arp::arp_receiver::
            receive, &layer_arp, _1, _2));
117
118     device_id dev1 = 1;
119     {
120         auto dev = layer_mac.get_or_create_dev_data(dev1);
121         dev->unicast_address_add({{0x00,0x23,0x14,0x30,0xce,0x74}})
                ;
122
123         auto v4 = layer_ipv4.get_or_create_dev_data(dev1);
124         v4->local_address_add({{10, 0, 0, 10}});
125
126         auto v6 = layer_ipv6.get_or_create_dev_data(dev1);
127         v6->local_address_add({{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 1}}, 0x2); // All nodes
128         v6->local_address_add({{0xfe, 0x80, 0, 0, 0, 0, 0, 0x02,
                0x23, 0x14, 0xff, 0xfe, 0x30, 0xce, 0x74}}, 0x2); //
                Link local
129     }
130
131     layer_ipv4.get_routing_table().add_route({{{10,0,0,0}}, 8, true
            , {{0,0,0,0}}, dev1});
132
133     // Set up a signal handler to catch Ctrl-C, so we can terminate
            correctly
134     struct sigaction signal_handler_INT;
135     signal_handler_INT.sa_handler = [](int s) { printf("Terminating
            ...\n"); running = false; };
136     sigemptyset(&signal_handler_INT.sa_mask);
137     signal_handler_INT.sa_flags = 0;
```

```
138     sigaction(SIGINT, &signal_handler_INT, nullptr);
139
140     devices::raw::netdevice_raw dev("side_b", 1024);
141     layer_mac.set_hook_request(std::bind(&decltype(dev)::send, &dev
            , _1));
142     auto worker_func = [&](int k) {
143         int x = 0;
144         while(running) {
145             int maxpull = 10;
146             auto retval = dev.poll(maxpull);
147             for(auto& pkt : retval) {
148                 --maxpull;
149                 ++x;
150                 pkt->dev = dev1;
151                 pkt->worker = k;
152                 layer_mac.receive(std::move(pkt));
153             }
154             if(maxpull != 0) {
155                 // Sleep if we did not receive a full batch of
                        packets.
156                 std::this_thread::sleep_for(std::chrono::
                        milliseconds(0));
157             }
158         }
159     };
160     std::vector<std::thread> workers;
161     for(auto i = 0; i < 2; ++i) {
162         workers.emplace_back(worker_func, i);
163     }
164     for(auto& w : workers) {
165         w.join();
166     }
167
168     return 0;
169 }
```

Listing C.12: trokis/containers/intrusive_queue_mpsc.hpp

```
1  //
2  //  intrusive_queue_mpsc.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 15/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_intrusive_queue_mpsc_hpp
10 #define trokis_intrusive_queue_mpsc_hpp
11
12 namespace trokis { namespace containers {
13
14     // Wait-free intrusive multiple-producer-single-consumer based
           on C version at:
15     //  http://www.1024cores.net/home/lock-free-algorithms/queues/
           intrusive-mpsc-node-based-queue
```

```
16      // Algorithm is unchanged from the C based version, but the
            interface is changed for better
17      // safety utilizing C++11 unique_ptr's. The given typename T
            must have at least a next member:
18      //  T* next
19      // The push interface accepts unique_ptr's and standard
            pointers. Only movable unique_ptr's are
20      // allowed, meaning the queue takes over ownership of the
            unique_ptr's object.
21      // The pop interface should only be called in single-threaded
            context and always returns
22      // unique_ptr's.
23      template<typename T>
24      class queue_mpsc {
25      public:
26          using value_type = T;
27
28          queue_mpsc() : head{&stub}, tail{&stub} {
29              stub.next = nullptr;
30          }
31
32          ~queue_mpsc() {
33              auto item = pop();
34              while(item) {
35                  item = pop();
36              }
37          }
38
39          // Takes control of a unique_ptr and adds the pointed to
                element to the list.
40          void push(std::unique_ptr<value_type>&& elem) {
41              auto tmp = std::move(elem);
42              push(tmp.release());
43          }
44
45          // Adds an element to the list in a wait-free manner.
46          void push(value_type* elem) {
47              if(elem == nullptr) {
48                  return;
49              }
50              elem->next = nullptr;
51              auto prev = head.exchange(elem);
52              // Note that a producer blocking right here will block
                    the consumer from seeing more items.
53              prev->next = elem;
54          }
55
56          // Returns the next item from the list.
57          std::unique_ptr<value_type> pop() {
58              auto tail_node = tail.load();
59              auto next_node = tail_node->next;
60              if(tail_node == &stub) {
61                  if(next_node == nullptr) {
62                      return nullptr;
63                  }
```

```
64                  tail.store(next_node);
65                  tail_node = next_node;
66                  next_node = next_node->next;
67              }
68              if(next_node != nullptr) {
69                  tail.store(next_node);
70                  return std::unique_ptr<value_type>(tail_node);
71              }
72              auto head_node = head.load();
73              if(tail_node != head_node) {
74                  return nullptr;
75              }
76              push(&stub);
77              next_node = tail_node->next;
78              if(next_node != nullptr) {
79                  tail.store(next_node);
80                  return std::unique_ptr<value_type>(tail_node);
81              }
82              return nullptr;
83          }
84
85      private:
86          value_type                 stub;
87          std::atomic<value_type*> head;
88          std::atomic<value_type*> tail;
89      };
90
91  } }
92
93  #endif
```

Listing C.13: trokis/detail/compat.h

```
1   //
2   //  compat.h
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 15/06/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_compat_h
10  #define trokis_compat_h
11
12  #include <cstddef>
13  #include <memory>
14  #include <type_traits>
15  #include <utility>
16
17  namespace std {
18      template<class T> struct _Never_true : false_type { };
19
20      template<class T> struct _Unique_if {
21          typedef unique_ptr<T> _Single;
22      };
```

```
23
24      template<class T> struct _Unique_if<T[]> {
25          typedef unique_ptr<T[]> _Runtime;
26      };
27
28      template<class T, size_t N> struct _Unique_if<T[N]> {
29          static_assert(_Never_true<T>::value, "make_unique forbids T
                [N]. Please use T[].");
30      };
31
32      template<class T, class... Args> typename _Unique_if<T>::
            _Single make_unique(Args&&... args) {
33          return unique_ptr<T>(new T(std::forward<Args>(args)...));
34      }
35
36      template<class T> typename _Unique_if<T>::_Single
            make_unique_default_init() {
37          return unique_ptr<T>(new T);
38      }
39
40      template<class T> typename _Unique_if<T>::_Runtime make_unique(
            size_t n) {
41          typedef typename remove_extent<T>::type U;
42          return unique_ptr<T>(new U[n]());
43      }
44
45      template<class T> typename _Unique_if<T>::_Runtime
            make_unique_default_init(size_t n) {
46          typedef typename remove_extent<T>::type U;
47          return unique_ptr<T>(new U[n]);
48      }
49
50      template<class T, class... Args> typename _Unique_if<T>::
            _Runtime make_unique_value_init(size_t n, Args&&... args) {
51          typedef typename remove_extent<T>::type U;
52          return unique_ptr<T>(new U[n]{ std::forward<Args>(args)...
                });
53      }
54
55      template<class T, class... Args> typename _Unique_if<T>::
            _Runtime make_unique_auto_size(Args&&... args) {
56          typedef typename remove_extent<T>::type U;
57          return unique_ptr<T>(new U[sizeof...(Args)]{ std::forward<
                Args>(args)... });
58      }
59  }
60
61  #endif
```

Listing C.14: trokis/devices/raw/netdevice_raw.hpp

```
1  //
2  //  netdevice_raw.hpp
3  //  trokis
4  //
```

```
5  //  Created by Lasse Bang Dalegaard on 18/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_netdevice_raw_hpp
10 #define trokis_netdevice_raw_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/support/dispatcher.hpp"
14
15 #include <string>
16 #include <vector>
17 #include <exception>
18 #include <cstring>
19 #include <fstream>
20
21 #include <sys/socket.h>
22 #include <sys/types.h>
23 #include <net/ethernet.h>
24 #include <net/if.h>
25 #include <netinet/in.h>
26 #include <linux/if_ether.h>
27 #include <netpacket/packet.h>
28 #include <unistd.h>
29 #include <fcntl.h>
30
31 namespace trokis { namespace devices { namespace raw {
32
33     namespace detail {
34
35         std::size_t interface_get_mtu(std::string devname) {
36             auto filename = std::string("/sys/class/net/") +
                    devname + std::string("/mtu");
37             std::ifstream input(filename);
38             std::size_t mtu;
39             input >> mtu;
40             return mtu;
41         }
42
43     }
44
45     // Implements a networking device that uses the Linux raw
            socket interface to send and receive
46     // raw ethernet frames to the network. This device is not multi
        -queue capable, but does support
47     // receiving from multiple concurrent threads, but data may be
            out of order in this case.
48     class netdevice_raw {
49     public:
50         netdevice_raw(std::string device_name, std::size_t backlog)
                : sock_(0) {
51             sock_ = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
52             if(sock_ == -1) {
53                 throw std::system_error(std::error_code(errno, std
                        ::system_category()), "Could not create socket"
```

```
                );
54          }
55
56          // Get interface MTU
57          mtu_ = detail::interface_get_mtu(device_name) + 14; //
                +14 bytes for ethernet header
58
59          // Find the interface index
60          ifindex_ = if_nametoindex(device_name.c_str());
61          if(ifindex_ == 0) {
62              throw std::runtime_error(std::string("Could not
                    find interface with name ") + device_name);
63          }
64
65          // Set up the sockaddr descriptor
66
67          sockaddr_ll lladdr;
68          memset(&lladdr, 0, sizeof(lladdr));
69          lladdr.sll_family = AF_PACKET;
70          lladdr.sll_protocol = htons(ETH_P_ALL);
71          lladdr.sll_ifindex = ifindex_;
72
73          // Bind to device
74          if(bind(sock_, reinterpret_cast<sockaddr*>(&lladdr),
                sizeof(lladdr)) != 0) {
75              throw std::system_error(std::error_code(errno, std
                    ::system_category()), "Could not bind to
                    interface");
76          }
77
78          // Set receive timeout
79
80          // Set non-blocking
81          if(fcntl(sock_, F_SETFL, O_NONBLOCK) != 0) {
82              throw std::system_error(std::error_code(errno, std
                    ::system_category()), "Could not set non-
                    blocking");
83          }
84
85          // Set options
86          packet_mreq req;
87          memset(&req, 0, sizeof(req));
88          req.mr_ifindex = ifindex_;
89          req.mr_type = PACKET_MR_PROMISC;
90          if(setsockopt(sock_, SOL_SOCKET, PACKET_ADD_MEMBERSHIP,
                &req, sizeof(req)) != 0) {
91              throw std::system_error(std::error_code(errno, std
                    ::system_category()), "Could not set promisc
                    mode");
92          }
93          memset(&req, 0, sizeof(req));
94          req.mr_ifindex = ifindex_;
95          req.mr_type = PACKET_MR_ALLMULTI;
96          if(setsockopt(sock_, SOL_SOCKET, PACKET_ADD_MEMBERSHIP,
                &req, sizeof(req)) != 0) {
```

```
97                  throw std::system_error(std::error_code(errno, std
                        ::system_category()), "Could not set all
                        multicast mode");
98              }
99          }
100         ~netdevice_raw() {
101             if(sock_ != 0) {
102                 close(sock_);
103             }
104         }
105
106         // Receives fully cooked ethernet packets from the upper
                layer protocols and
107         // forwards them on to the wire.
108         packet_verdict send(packet_ptr&& pkt) {
109             auto& pkt_data = pkt->data();
110             if(!pkt_data.may_pull(12)) return packet_verdict::
                    dropped;
111             auto segments = pkt_data.raw_segments();
112             std::vector<iovec> iov(segments.size());
113             transform(std::begin(segments), std::end(segments), std
                    ::begin(iov), [](const std::pair<byte_t*,std::
                    size_t>& v) {
114                 iovec retval;
115                 retval.iov_base = v.first;
116                 retval.iov_len = v.second;
117                 return retval;
118             });
119
120             sockaddr_ll lladdr;
121             memset(&lladdr, 0, sizeof(lladdr));
122             lladdr.sll_family = AF_PACKET;
123             lladdr.sll_halen = 6;
124             lladdr.sll_ifindex = ifindex_;
125             std::copy_n(pkt_data.begin() + 6, 6, lladdr.sll_addr);
126
127             msghdr msg;
128             memset(&msg, 0, sizeof(msg));
129             msg.msg_iov = iov.data();
130             msg.msg_iovlen = iov.size();
131             msg.msg_name = &lladdr;
132             msg.msg_namelen = sizeof(lladdr);
133             auto retval = sendmsg(sock_, &msg, 0);
134             if(retval < 0) {
135                 throw std::system_error(std::error_code(errno, std
                        ::system_category()), "Sending");
136             }
137             return (retval > 0) ? packet_verdict::success :
                    packet_verdict::dropped;
138         }
139
140         // Polls the ethernet device for packets and returns a
                vector of packets received.
141         std::vector<packet_ptr> poll(std::size_t max_packets) {
142             std::vector<packet_ptr> retval;
```

```
143             retval.reserve(max_packets);
144             for(std::size_t i = 0; i < max_packets; ++i) {
145                 // Allocate data
146                 std::vector<byte_t> buffer(mtu_);
147 retry:
148                 iovec iov;
149                 iov.iov_base = buffer.data();
150                 iov.iov_len  = buffer.size();
151                 sockaddr_ll lladdr;
152                 memset(&lladdr, 0, sizeof(lladdr));
153                 msghdr msg;
154                 memset(&msg, 0, sizeof(msg));
155                 msg.msg_iov = &iov;
156                 msg.msg_iovlen = 1;
157                 msg.msg_name = &lladdr;
158                 msg.msg_namelen = sizeof(lladdr);
159                 auto length = recvmsg(sock_, &msg, 0);
160                 if(length == 0) {
161                     // We will never receive a packet again
162                     break;
163                 }
164                 if(length < 0) {
165                     if(errno == EAGAIN || errno == EWOULDBLOCK) {
166                         // If the socket would have blocked, no
                                more packets can be pulled out. Return.
167                         break;
168                     }
169                     // Some unknown error message. We'll just throw
                            an exception.
170                     throw std::system_error(std::error_code(errno,
                            std::system_category()), "Error while
                            receiving");
171                 }
172                 if(lladdr.sll_pkttype == PACKET_OUTGOING) goto
                        retry;
173                 // A packet was received
174                 buffer.resize(length);
175                 auto data_ptr = std::make_unique<packet_data>(std::
                        move(buffer));
176                 auto ptr = std::make_unique<packet>();
177                 ptr->replace_data(std::move(data_ptr));
178                 retval.push_back(std::move(ptr));
179             }
180             return retval;
181         }
182
183     private:
184         std::size_t mtu_;
185         int sock_;
186         unsigned int ifindex_;
187     };
188
189 } } }
190
191 #endif
```

Listing C.15: trokis/neigh_provider/arp.hpp

```
1  //
2  //  arp_provider.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 15/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_arp_provider_hpp
10 #define trokis_arp_provider_hpp
11
12 #include "trokis/support/neighbor_table.hpp"
13
14 #include <functional>
15
16 namespace trokis { namespace neighbor {
17
18     // Implements ARP transmission, currently only for IP over
       ethernet but can of course be extended to any
19     // required protocol pair.
20     template<typename ProtocolType, typename HardwareInfo>
21     class arp_provider {
22     public:
23         using neighbor_type = neighbor::generic_neighbor<
           arp_provider>;
24         using upper_address_type = ProtocolType;
25         using lower_address_type = typename HardwareInfo::
           address_type;
26
27         using output_hook_type = std::function<packet_verdict(
           packet_ptr&&, lower_address_type, lower_address_type)>;
28         using primary_address_hook_type = std::function<std::pair<
           upper_address_type,lower_address_type>(device_id)>;
29
30         arp_provider()
31         : output_hook_([](packet_ptr&&, lower_address_type,
           lower_address_type) { return packet_verdict::dropped;
           })
32         , primary_address_hook_([](device_id) { return std::
           make_pair(upper_address_type(), lower_address_type());
           })
33         { }
34
35         template<typename Func>
36         void set_output_hook(Func&& f) {
37             output_hook_ = std::forward<Func>(f);
38         }
39
40         template<typename Func>
41         void set_primary_address_hook(Func&& f) {
42             primary_address_hook_ = std::forward<Func>(f);
43         }
44
```

```
45          // FUTURE: Update this so that other protocols are also
                supported, currently this only works if
46          // HardwareType = ieee802::mac_address and ProtocolType =
                ipv4::address (or equivalents)
47          void solicit(device_id devid, const upper_address_type&
                addr) {
48              // Create an ARP packet and send it
49              auto pkt = std::make_unique<packet>();
50              pkt->dev = devid;
51              auto& pkt_data = pkt->data();
52              pkt_data.reserve_front(64); // Reserve 64 bytes in
                    front to help out the lower levels.
53              pkt_data.put(28); // ARP with IP <-> ethernet is a 28
                    byte payload
54              pkt_data.set_bytes_be(+0, (uint16_t)1);          //
                    Ethernet has HTYPE = 1
55              pkt_data.set_bytes_be(+2, (uint16_t)0x0800);     // IP
                    has PTYPE = 0x0800, same as ethertype
56              pkt_data.set_byte(+4, 6);                        // 6
                    octets in a MAC address
57              pkt_data.set_byte(+5, 4);                        // 4
                    octets in an IP address
58              pkt_data.set_bytes_be(+6, (uint16_t)1);          // ARP
                    request has operation = 1
59
60              auto primary_addresses = primary_address_hook_(devid);
61
62              // Sender(that is our) address is the "primary" address
                     from the netdevice
63              pkt_data.set_bytes(+8, std::get<1>(primary_addresses));
64
65              // Our protocol address is our (primary) IP address
66              pkt_data.set_bytes(+14, std::get<0>(primary_addresses))
                    ;
67
68              // We don't know the target hardware address, so we
                    simply fill it with the ethernet broadcast address
69              pkt_data.set_bytes(+18, HardwareInfo::
                    addr_global_broadcast);
70
71              // Finally the IP we are looking for
72              pkt_data.set_bytes(+24, addr);
73
74              // Push an ethertype on, ARP is 0x0806
75              pkt->data().push(2);
76              pkt->data().set_bytes_be(+0, (uint16_t)0x0806);
77
78              // Send it out. We don't care about the result of the
                    transmission itself.
79              raw_output(std::move(pkt), std::get<1>(
                    primary_addresses), HardwareInfo::
                    addr_global_broadcast);
80          }
81
82          // Performs raw output to the output hook.
```

```
83          packet_verdict raw_output(packet_ptr&& pkt, const
                lower_address_type& src, const lower_address_type& dst)
                 {
84              return output_hook_(std::move(pkt), src, dst);
85          }
86
87          // Encapsulates a frame and sends it to the output hook.
                The frame is encapsulated
88          // as an IPv4 frame, and assumes that the lower level is
                IEEE 802.3. Must be extended
89          // for full LLC support(and other protocols) later on.
90          packet_verdict output(packet_ptr&& pkt, const
                lower_address_type& src, const lower_address_type& dst)
                 {
91              pkt->data().push(2);
92              pkt->data().set_bytes_be(+0, (uint16_t)0x0800);
93              return raw_output(std::move(pkt), src, dst);
94          }
95
96      private:
97          output_hook_type output_hook_;
98          primary_address_hook_type primary_address_hook_;
99      };
100
101 } }
102
103 #endif
```

Listing C.16: trokis/packet.hpp

```
1  //
2  //  packet.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_packet_hpp
10 #define trokis_packet_hpp
11
12 #include "trokis/detail/compat.h"
13 #include "trokis/packet_data.hpp"
14
15 #include <memory>
16
17 namespace trokis {
18
19     // Wrapper around a packet. Basically a POD type containing
            high-level data about
20     // the packet. Also uniques a pointer to the actual payload
            data of the packet.
21     // The packet type performs no locking of any kind, as only a
            single thread is ever
```

```
22      // supposed to work with a given packet instance at any one
            time. In the client code,
23      // packet instances are always wrapped in packet_ptr objects.
24      class packet {
25      public:
26          packet() {
27              data_ = std::make_unique<packet_data>();
28          }
29
30          // Retrieves the payload data area
31          packet_data& data() const {
32              return *data_;
33          }
34
35          void replace_data(std::unique_ptr<packet_data>&& newdata) {
36              data_ = std::move(newdata);
37          }
38
39          // Device references
40          device_id dev; // "Current" device
41
42          // Worker that is handling this packet
43          worker_id worker;
44
45      private:
46          std::unique_ptr<packet_data> data_;
47      };
48
49  }
50
51  #endif
```

Listing C.17: trokis/packet_data.hpp

```
1   //
2   //  packet_data.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 09/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef TROKIS_PACKET_DATA
10  #define TROKIS_PACKET_DATA
11
12  #include "trokis/types.hpp"
13  #include "trokis/support/byte_tools.hpp"
14
15  #include <vector>
16  #include <cassert>
17
18  namespace trokis {
19
20      // Implements an example trokis packet_data backend. The
            packet_data structure
```

```
21      // implemenets a generic model for payload handling in the
            trokis network stack. Packet data
22      // is represented as an array of bytes with extra indexing to
            manage excess capacity.
23      // Conceptually, the data in a packet can be visualized as
            follows:
24      //
25      // (Begin)       Head                              Tail         (End)
26      //    |------------|--------------------------------|------------|
27      //    | Head room  |            Payload             | Tail room  |
28      //    |------------|--------------------------------|------------|
29      //
30      // This model is very similar to the end used in Linux, FreeBSD
            and others. From the users
31      // point of view, only the payload area is valid, and the head
            and tail rooms can be considered
32      // infinite. The API provides functionality for moving the head
            and tail pointers, and also
33      // provides functionality to read and write to the Payload area
            . All data access is relative to
34      // the Head pointer.
35      //
36      // Bytes can be read and written to using the get_* and set_
            functions, and direct access to the
37      // bytes in the payload area are available using the STL-like
            begin() and end() functions. These
38      // return iterators, allowing standard STL algorithms to be
            used with the container. The iterator
39      // returned by begin() points to the Head location while the
            iterator returned by end() points to
40      // the first location past Tail.
41      //
42      // The head pointer can be moved forward(payload size decreased
            ) using the pull(...) function,
43      // and backward(payload size increased) using the push(...)
            function. Both functions take the
44      // number of bytes to move the head pointer.
45      //
46      // Similarly, the tail pointer can be moved backward(payload
            size decreased) using trim(...),
47      // and forward(payload size increased) using put(...).
48      //
49      // Finally, head room can be reserved using the reserve_head
            function while the reserve_capacity
50      // function can be used to reserve a total length of the buffer
            . Space reservation should be
51      // considered a hint by the client.
52      //
53      // It is undefined if the iterators returned by begin() and end
            () are invalidated by Head/Tail
54      // movement, but the client should assume that they will be
            invalidated. Likewise, space reservation
55      // may invalidate all iterators.
56      //
```

```
57      // This initial implementation uses a simple std::vector as the
            backend buffer. This allows fast
58      // access to data because of the continuous layout in memory,
            but of course moving the Head or Tail
59      // pointer out of the current vector will result in
            reallocation of the memory buffer, forcing a
60      // full copy of all data. A future implementation could support
            fragmented data buffers, at the
61      // expense of somewhat more expensive get_* and set_* helpers(
            along with iterators) as these can no
62      // longer assume continous memory. The packet_data abstraction
            exists exactly to remove this detail
63      // from the client code.
64      //
65      // For transmitting, we provide the raw_segments function,
            allowing the client to get a list of raw
66      // data areas that should be combined to form the complete
            packet on outputting.
67      //
68      class packet_data {
69      public:
70          using backing_type = std::vector<byte_t>;
71
72          // In liu of inherited constructors, we can use this neat
                perfect forwarding trick
73          template<typename... Args>
74          packet_data(Args&&... args) : backing_(std::forward<Args>(
                args)...), head_(0) { }
75
76          packet_data() : backing_(), head_(0) { }
77
78          // Various attribute getters
79          std::size_t length() const {
80              return backing_.size() - head_;
81          }
82
83          std::size_t capacity() const {
84              return backing_.capacity();
85          }
86
87          // Reserve in front of head
88          void reserve_front(std::ptrdiff_t count) {
89              assert(backing_.size() >= head_);
90              if(length() == 0) {
91                  // No need to actually move data, but we need to
                        reserve at least this capacity
92                  backing_.resize(backing_.size() + count);
93              } else {
94                  backing_.insert(backing_.begin(), count, 0);
95              }
96              head_ += count;
97              assert(backing_.size() >= head_);
98          }
99
100         void reserve_length(std::ptrdiff_t count) {
```

```
101             backing_.reserve(head_ + count);
102         }
103
104         // Head adjustment
105         void push(std::size_t count) {
106             if(count > head_) {
107                 backing_.insert(backing_.begin(), count, 0);
108             } else {
109                 head_ -= count;
110             }
111         }
112
113         void pull(std::size_t count) {
114             head_ += count;
115             if(head_ > backing_.size()) {
116                 backing_.insert(backing_.end(), count, 0);
117             }
118         }
119
120         // Determine if we can read this amount of bytes
121         bool may_pull(std::size_t count) {
122             return length() >= count;
123         }
124
125         // Tail adjustment
126         void put(std::size_t count) {
127             backing_.resize(backing_.size() + count);
128         }
129         void trim(std::size_t count) {
130             backing_.resize(backing_.size() - count);
131             if(backing_.size() < head_) {
132                 head_ = backing_.size();
133             }
134         }
135
136         // Get iterators
137         backing_type::iterator begin() {
138             return backing_.begin() + head_;
139         }
140
141         backing_type::iterator end() {
142             return backing_.end();
143         }
144
145         backing_type::const_iterator cbegin() {
146             return backing_.cbegin() + head_;
147         }
148
149         backing_type::const_iterator cend() {
150             return backing_.cend();
151         }
152
153         // Retrieves bytes
154         template<std::size_t N>
155         uint64_t get_bytes_be(std::ptrdiff_t offset) const {
```

```
156             return byte_tools::read_bytes<N, byte_tools::
                    endianness_big>(backing_.data() + head_ + offset);
157         }
158
159         template<std::size_t N>
160         uint64_t get_bytes_le(std::ptrdiff_t offset) const {
161             return byte_tools::read_bytes<N, byte_tools::
                    endianness_little>(backing_.data() + head_ + offset
                    );
162         }
163
164         template<std::size_t N>
165         std::array<byte_t, N> get_bytes(std::ptrdiff_t offset)
                const {
166             return byte_tools::read_bytes_array<N>(backing_.data()
                    + head_ + offset);
167         }
168
169         byte_t get_byte(std::ptrdiff_t offset) const {
170             return *(backing_.data() + head_ + offset);
171         }
172
173         // Puts bytes
174         template<typename T>
175         void set_bytes_be(std::ptrdiff_t offset, T&& data) {
176             byte_tools::write_bytes<byte_tools::endianness_big>
177             (std::forward<T>(data), backing_.data() + head_ +
                    offset);
178         }
179
180         template<typename T>
181         void set_bytes_le(std::ptrdiff_t offset, T data) {
182             byte_tools::write_bytes<byte_tools::endianness_little>
183             (std::forward<T>(data), backing_.data() + head_ +
                    offset);
184         }
185
186         template<std::size_t N>
187         void set_bytes(std::ptrdiff_t offset, const std::array<
                byte_t, N>& data) {
188             byte_tools::write_bytes_array<N>(data, backing_.data()
                    + head_ + offset);
189         }
190
191         void set_byte(std::ptrdiff_t offset, byte_t data) {
192             *(backing_.data() + head_ + offset) = data;
193         }
194
195         // Returns the segments of the packet. Since this is backed
                by a vector, we simply return a
196         // one-element vector with a single (pointer,length) pair.
197         std::vector<std::pair<byte_t*, std::size_t>> raw_segments()
                {
198             return {std::make_pair(backing_.data() + head_, length
                    ())};
```

```
199         }
200
201     private:
202         backing_type backing_;
203         std::size_t head_;
204     };
205
206 }
207
208 #endif
```

Listing C.18: trokis/protocols/ieee802/ieee802_3.hpp

```
1  //
2  //  ieee802_3.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_802_hpp
10 #define trokis_802_hpp
11
12 #include "trokis/protocols/ieee802/types.hpp"
13
14 #include <mutex>
15 #include <unordered_map>
16 #include <cassert>
17
18 namespace trokis { namespace ieee802 {
19
20     namespace detail {
21         template<typename Func, typename... Args>
22         auto encap_ethertype(Func&& f, uint16_t ethertype,
                packet_ptr&& pkt, Args&&... args)
23         -> typename std::result_of<Func(packet_ptr&&, Args...)>::
                type {
24             pkt->data().push(2);
25             pkt->data().set_bytes_be(+0, ethertype);
26             return f(std::move(pkt), std::forward<Args>(args)...);
27         }
28     }
29
30     // Encapsulates upper 802 sublayers, namely Ethernet and LLC
            sublayers.
31     // These sublayers are in the same class because they need to
            call eachother and still have
32     // access to the hook to transmit to the upper level.
33     // MAC sublayers are allowed to inherit from this, giving them
            the required functionality to
34     // use in their own hooks if they require it.
35     template<typename DeviceData>
36     struct top_sublayers_802 : public utilities::generic_block<
37         packet_verdict(packet_ptr&&, ethertype, address_type),
```

```
38          packet_verdict(packet_ptr&& pkt)
39      > {
40
41          packet_verdict ether_decap_llc(packet_ptr&& pkt,
                address_type linklayer_type) {
42              // No LLC support yet.
43              return packet_verdict::dropped;
44          }
45
46          // Ethernet sublayer decapsulation
47          packet_verdict ether_decap(packet_ptr&& pkt, address_type
                linklayer_type) {
48              // Drop if we can't pull out an ethertype
49              if(!pkt->data().may_pull(2)) {
50                  return packet_verdict::dropped;
51              }
52              auto ethertype = pkt->data().get_bytes_be<2>(+0);
53              pkt->data().pull(2);
54              if(ethertype <= 1500) {
55                  // LLC sublayer packet
56                  return ether_decap_llc(std::move(pkt),
                        linklayer_type);
57              } else if(ethertype >= 1536) {
58                  // Ethernet sublayer packet. We don't yet support
                        VLANs, so simply inform upper.
59                  return inform(std::move(pkt), ethertype,
                        linklayer_type);
60              } else {
61                  return packet_verdict::dropped;
62              }
63          }
64
65          template<typename... Args>
66          DeviceData* get_or_create_dev_data(device_id id, Args&&...
                args) {
67              std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
68              auto retval = dev_data_.emplace(id, gc::make_gc_ptr<
                    DeviceData>(std::forward<Args>(args)...));
69              return retval.first->second.acquire();
70          }
71
72          void drop_dev_data(device_id id) {
73              std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
74              // Note that the actual dev_data entry will be
                    reclaimed later by gc_ptr
75              dev_data_.erase(id);
76          }
77
78          DeviceData* get_dev_data(device_id id) {
79              std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
80              auto iter = dev_data_.find(id);
81              if(iter == dev_data_.end()) return nullptr;
```

```
82              return iter->second.acquire();
83          }
84
85      private:
86          std::mutex dev_data_lock_;
87          std::unordered_map<device_id, gc::gc_ptr<DeviceData>>
                dev_data_;
88      };
89
90      // Encapsulates 802.3 MAC
91      class ethernet_802_3 : public top_sublayers_802<
            dev_data_ieee802_3> {
92      public:
93          // Receives 802.3 MAC frames and removes their initial MAC
                address header.
94          packet_verdict receive(packet_ptr&& pkt) {
95              auto dev = get_dev_data(pkt->dev);
96              if(!dev) {
97                  return packet_verdict::dropped;
98              }
99              // Drop if the packet isn't at least 12 bytes(dst + src
                    addresses)
100             if(!pkt->data().may_pull(12)) {
101                 return packet_verdict::dropped;
102             }
103
104             mac_address dst = pkt->data().get_bytes<6>(+0);
105             // Forward all multicast, all broadcast and only the
                    local unicast to upper layers.
106             auto is_multicast = (dst[0] & 0x1) != 0;
107             if(dev->has_unicast_address(dst) || is_multicast) {
108                 auto is_broadcast = (dst == dev_data_ieee802_3::
                        info::addr_global_broadcast);
109                 address_type type = is_multicast ? address_type::
                        multicast :
110                     (is_broadcast ? address_type::broadcast :
                            address_type::unicast);
111                 pkt->data().pull(12);
112                 return ether_decap(std::move(pkt), type);
113             } else {
114                 // FUTURE: Implement multicast and bridging
115                 return packet_verdict::dropped; // No bridging
                        support yet
116             }
117         }
118
119         mac_address get_primary_address(device_id devid) {
120             auto dev_data = get_dev_data(devid);
121             if(!dev_data) return dev_data_ieee802_3::info::
                    addr_global_broadcast;
122             return dev_data->get_primary_address();
123         }
124
125         // Packet transmission path. The packet must have already
                been protocol encapsulated
```

```
126         // at this point.
127         packet_verdict transmit(packet_ptr&& pkt, mac_address src,
                mac_address dst) {
128             auto dev = get_dev_data(pkt->dev);
129             if(!dev) {
130                 return packet_verdict::dropped;
131             }
132
133             if(src == mac_address()) {
134                 src = dev->get_primary_address();
135             }
136             pkt->data().push(12);
137             pkt->data().set_bytes(+0, dst);
138             pkt->data().set_bytes(+6, src);
139
140             // If the total length is less than 64, pad with zero.
141             int padlength = 64 - pkt->data().length();
142             if(padlength > 0) {
143                 pkt->data().put(padlength);
144                 std::fill(pkt->data().end() - padlength, pkt->data
                        ().end(), 0);
145             }
146
147             return request(std::move(pkt));
148         }
149     };
150
151 } }
152
153 #endif
```

Listing C.19: trokis/protocols/ieee802/types.hpp

```
1  //
2  //  ieee802_types.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 13/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ieee802_types_hpp
10 #define trokis_ieee802_types_hpp
11
12 #include "trokis/support/neighbor_table.hpp"
13 #include "trokis/neigh_provider/arp.hpp"
14
15 #include <array>
16
17 namespace trokis { namespace ieee802 {
18
19     // An IEEE 802 48-bit MAC address
20     // using mac_address = std::array<byte_t, 6>;
21
22     using mac_address = std::array<byte_t, 6>;
```

```
23
24      struct dev_info_802_3 {
25          static mac_address addr_global_broadcast;
26          using address_type = mac_address;
27      };
28      mac_address dev_info_802_3::addr_global_broadcast = {{0xFF, 0
            xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
29
30      // Ethertype field used in the Ethernet sublayer.
31      using ethertype = uint16_t;
32
33      // Describes an 802.3 networking device, ie. ethernet. Provides
             helper functions for managing
34      // addresses and configuration data of the device.
35      class dev_data_ieee802_3 {
36      public:
37          using info = dev_info_802_3;
38
39          dev_data_ieee802_3()
40          : unicast_addresses_(gc::make_gc_ptr<decltype(
                unicast_addresses_)::element_type>())
41          { }
42
43          // Helper: CAS updates the list of unicast addresses.
44          void unicast_address_add(mac_address addr) {
45              unicast_addresses_.update([&](decltype(
                    unicast_addresses_)::element_type& list) {
46                  list.push_back(addr);
47              });
48          }
49
50          // Helper: Tests if the interface has the specified address
                 as a unicast address
51          bool has_unicast_address(mac_address addr) {
52              auto& current_list = *unicast_addresses_;
53              auto retval = std::find(std::begin(current_list), std::
                    end(current_list), addr);
54              return retval != std::end(current_list);
55          }
56
57          // Helper: Retrieves the primary(first) address of the
                interface
58          mac_address get_primary_address() {
59              auto& current_list = *unicast_addresses_;
60              return current_list.size() != 0 ? *current_list.begin()
                     : info::addr_global_broadcast;
61          }
62
63      private:
64          gc::gc_ptr<std::vector<mac_address>> unicast_addresses_;
65      };
66
67  } }
68
69  #endif
```

Listing C.20: trokis/protocols/ipv4/arp.hpp

```
1  //
2  //  arp.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 14/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_arp_hpp
10 #define trokis_arp_hpp
11
12 #include "trokis/protocols/ipv4/types.hpp"
13 #include "trokis/protocols/ieee802/types.hpp"
14
15 namespace trokis { namespace arp {
16
17     // Receives and parses ARP packets. Only ethernet/IP is
       supported at the moment.
18     // Inputs:
19     //  packet_verdict receive(packet_ptr&&, mac_address,
       mac_address)
20     // Output hooks:
21     //  packet_verdict request(packet_ptr&&, mac_address)
22     // Data hooks:
23     //  bool update_neighbor(device_id, ip_address nexthop,
       mac_address newaddr, bool may_create)
24     //  bool is_local_address(device_id, ip_address address)
25     //
26     class arp_receiver : public utilities::
           generic_block_request_hook<
27         packet_verdict(packet_ptr&&, ieee802::mac_address, ieee802
               ::mac_address)
28     > {
29     public:
30         using update_neighbor_hook_t = std::function<bool(device_id
               , ipv4::address_t, ieee802::mac_address, bool)>;
31         using is_local_address_hook_t = std::function<bool(
               device_id, ipv4::address_t)>;
32         using primary_address_hook_t = std::function<ieee802::
               mac_address(device_id)>;
33
34         arp_receiver()
35         : update_neighbor_hook_([](device_id, ipv4::address_t,
               ieee802::mac_address, bool) { return false; })
36         , is_local_address_hook_([](device_id, ipv4::address_t) {
               return false; })
37         , primary_address_hook_([](device_id) { return ieee802::
               mac_address(); })
38         {};
39
40         packet_verdict receive(packet_ptr&& pkt, address_type
               linklayer_type) {
41             // If this device has no neighbor table, it shouldn't
                   support ARP so drop packet
```

```
42              if(pkt->data().length() < 28) { // Size of an ARP
                    packet with HTYPE = ethernet and PTYPE = IP
43                  return packet_verdict::dropped;
44              }
45              auto& pkt_data = pkt->data();
46
47              auto htype = pkt_data.get_bytes_be<2>(+0);
48              auto ptype = pkt_data.get_bytes_be<2>(+2);
49              if(htype != 1 || ptype != 0x0800) {
50                  // Ignore ARP packets that aren't for IP over
                        ethernet
51                  return packet_verdict::dropped;
52              }
53              if(pkt_data.get_byte(+4) != 6 || pkt_data.get_byte(+5)
                    != 4) {
54                  // Ignore ARP packet as the lengths don't match
55                  return packet_verdict::dropped;
56              }
57
58              // Read addresses
59              auto sha = pkt_data.get_bytes<6>(+8);   // Sender
                    Hardware Address
60              auto spa = pkt_data.get_bytes<4>(+14);  // Sender
                    Protocol Address
61              auto tpa = pkt_data.get_bytes<4>(+24);  // Target
                    Protocol Address
62
63              // Read the operation field
64              auto operation = pkt_data.get_bytes_be<2>(+6);
65
66              // Follow RFC826 packet reception section
67              bool merge_flag = false;
68              if(update_neighbor_hook_(pkt->dev, spa, sha, false)) {
69                  merge_flag = true;
70              }
71              if(is_local_address_hook_(pkt->dev, tpa)) {
72                  if(!merge_flag) {
73                      update_neighbor_hook_(pkt->dev, spa, sha, true)
                            ;
74                  }
75                  if(operation == 1) { // Message is a request and we
                        need to reply
76                      // Set the operation as REPLY = 2
77                      pkt_data.set_bytes_be(+6, (uint16_t)2);
78
79                      // Swap sender and target addresses
80                      std::swap_ranges(pkt_data.begin() + 8, pkt_data
                            .begin() + 18,
81                                       pkt_data.begin() + 18);
82
83                      // Fill in a new sender address(our own address
                            )
84                      auto host_address = primary_address_hook_(pkt->
                            dev);
85                      pkt_data.set_bytes(+8, host_address);
```

```
86                      return request(std::move(pkt), host_address,
                            sha);
87                  }
88              }
89              return packet_verdict::success;
90          }
91
92          template<typename Func>
93          void set_update_neighbor_hook(Func&& f) {
94              update_neighbor_hook_ = std::forward<Func>(f);
95          }
96
97          template<typename Func>
98          void set_is_local_address_hook(Func&& f) {
99              is_local_address_hook_ = std::forward<Func>(f);
100         }
101
102         template<typename Func>
103         void set_primary_address_hook(Func&& f) {
104             primary_address_hook_ = std::forward<Func>(f);
105         }
106
107     private:
108         update_neighbor_hook_t update_neighbor_hook_;
109         is_local_address_hook_t is_local_address_hook_;
110         primary_address_hook_t primary_address_hook_;
111     };
112
113 } }
114
115 #endif
```

Listing C.21: trokis/protocols/ipv4/defragmenter.hpp

```
1  //
2  //  defragmenter.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 17/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_defragmenter_hpp
10 #define trokis_defragmenter_hpp
11
12 #include "trokis/protocols/ipv4/types.hpp"
13
14 #include <tuple>
15 #include <chrono>
16 #include <mutex>
17 #include <vector>
18 #include <algorithm>
19
20 namespace trokis { namespace ipv4 {
21
```

```
22      // IPv4 packet defragmentation module
23      class defragmenter {
24      private:
25          // Tuple of src, dst, protocol and ID fields
26          using buffer_identifier = std::tuple<device_id, address_t,
                address_t, protonum_t, uint16_t>;
27          // Tuple of global offset, local offset, length and actual
                packet data to copy into the fragment. If
28          // packet_ptr is an empty pointer, this represents an empty
                span.
29          using fragment_area = std::tuple<std::size_t, std::size_t,
                std::size_t, packet_ptr>;
30
31          using fragment_clock = std::chrono::steady_clock;
32
33          // Holds a single fragment buffer. A list of packets is
                held, along with meta data to reassmeble the
34          // packet and also a timestamp to allow pruning the table
                later on.
35          struct fragment_buffer {
36              fragment_buffer()
37              : header_length(0), last_time(fragment_clock::now()),
                    total_length(0), buffered_count(0) { }
38
39              void reset() {
40                  header_length = 0;
41                  data_buffer.clear();
42                  total_length = 0;
43                  buffered_count = 0;
44              }
45
46              std::mutex lock;                                // Lock to
                    protect this fragment buffer
47              std::array<byte_t, 4 * 15>  header_buffer;  // Buffer
                    to contain header of final packet
48              std::size_t                 header_length;  // Length
                    of the header stored in header_buf
49              std::vector<fragment_area>  data_buffer;    // List of
                    fragment areas for the final fragment.
50                                                          // Note
                                                                that we
                                                                 can
                                                                use a
                                                                vector
                                                                since
                                                                we are
                                                                under
                                                                lock
51              fragment_clock::time_point  last_time;      // Time
                    when a fragment was last received
52              std::size_t                 total_length;   // Total
                    length of the packet
53              std::size_t                 buffered_count; // Number
                    of bytes of the total length that we have buffered
54          };
```

```
55
56      public:
57          defragmenter() { }
58          defragmenter(defragmenter&& other) {
59              std::unique_lock<decltype(frag_buffer_lock_)> guard(
                    other.frag_buffer_lock_);
60              frag_buffers_ = std::move(other.frag_buffers_);
61          }
62
63          // Receives a fragment of an IPv4 packet and performs all
                actions required to reassemble the packet.
64          // If the packet was reassembled, the resulting reassembled
                 packet is returned. If the packet could
65          // not be reassembled yet, an empty packet_ptr is returned.
66          packet_ptr fragment_receive(packet_ptr&& pkt, size_t
                header_length, protonum_t protocol,
67                                      address_t src, address_t dst) {
68              auto& pkt_data = pkt->data();
69              // Read packet identifier
70              auto pkt_id = pkt_data.get_bytes_be<2>(+4);
71
72              auto workerid = pkt->worker;
73              auto devid = pkt->dev;
74
75              // Determine if there are more fragments
76              auto pkt_flags = pkt_data.get_byte(+6);
77              bool more_fragments = pkt_flags & 0x20;
78
79              // Determine if this is the first fragment
80              auto fragment_offset = pkt_data.get_bytes_be<2>(+6) &
                    (0x1FFF);
81              bool is_first = fragment_offset == 0;
82
83              // This is not a fragmented packet!
84              if(is_first && !more_fragments) {
85                  return std::move(pkt);
86              }
87
88              auto offset = fragment_offset * 8;
89              // The packet already arrived trimmed for us, meaning
                    we can simply get the length from
90              // packet data and subtract the header_length we were
                    supplied.
91              auto payload_length = pkt_data.length() - header_length
                    ;
92
93              auto id = std::make_tuple(devid, src, dst, protocol,
                    pkt_id);
94              decltype(frag_buffers_)::iterator ibuffer;
95              fragment_buffer* buffer;
96              { // Critical section getting the fragment_buffer
97                  std::unique_lock<decltype(frag_buffer_lock_)> guard
                        (frag_buffer_lock_);
98                  ibuffer = frag_buffers_.find(id);
99                  if(ibuffer == frag_buffers_.end()) {
```

```
100                     auto retval = frag_buffers_.emplace(id, gc::
                            make_gc_ptr<fragment_buffer>());
101                     ibuffer = retval.first;
102                 }
103                 buffer = ibuffer->second.acquire();
104             }
105             { // Critical section performing the defragmentation
106                 std::unique_lock<decltype(fragment_buffer::lock)>
                        guard(buffer->lock);
107                 buffer->last_time = fragment_clock::now();
108                 // If this is the first packet, we have to set the
                        header
109                 if(more_fragments && is_first) {
110                     std::copy(pkt_data.begin(), pkt_data.begin() +
                            header_length, std::begin(buffer->
                            header_buffer));
111                     buffer->header_length = header_length;
112                 }
113
114                 // If this is the last fragment, we can set the
                        total length of the new packet.
115                 if(!more_fragments && !is_first) {
116                     buffer->total_length = offset + payload_length;
117                 }
118
119                 // Insert the new fragment
120                 buffer->data_buffer.emplace_back(offset,
                        header_length, payload_length, std::move(pkt));
121
122                 // Account for received data
123                 buffer->buffered_count += payload_length;
124                 if((buffer->buffered_count + buffer->header_length)
                         > 65535) {
125                     // If the amount of data received exceeds the
                            amount that can fit in an IPv4 packet,
126                     // cancel reassembly.
127                     buffer->reset();
128                     return nullptr;
129                 }
130
131                 if(buffer->total_length != 0 && buffer->
                        total_length <= buffer->buffered_count) {
132                     if(buffer->total_length < buffer->
                            buffered_count || buffer->header_length ==
                            0) {
133                         // Someone sent us nasty data. To be safe,
                                drop everything.
134                         buffer->reset();
135                         return nullptr;
136                     }
137                     // We have received the entire packet, so now
                            we can reassemble it!
138                     // First we sort by incresing packet offset.
139                     std::sort(std::begin(buffer->data_buffer), std
                            ::end(buffer->data_buffer),
```

```
140                 [](const fragment_area& a, const
                        fragment_area& b) {
141                     return std::get<0>(a) < std::get
                            <0>(b);
142                 });
143             // FUTURE: Change this to reuse a packet or
                    even better, combine the old packets. For
                    now,
144             // allocate an entirely new packet and fill it
                    in. Since this data is going up the stack,
145             // we don't generally need to reserve head room
                     as the packets are being decapsulated. The
146             // overhead of allocation an entirely new
                    packet should be small, and the code is
                    must
147             // simpler.
148             auto newpkt = std::make_unique<packet>();
149             newpkt->worker = workerid;
150             newpkt->dev = devid;
151             auto& newpkt_data = newpkt->data();
152             newpkt_data.put(buffer->header_length + buffer
                    ->total_length);
153             std::copy(std::begin(buffer->header_buffer),
                    std::end(buffer->header_buffer),
154                       newpkt_data.begin());
155             std::size_t expected_offset = buffer->
                    header_length;
156             for(auto& fragment : buffer->data_buffer) {
157                 std::size_t global_offset = std::get<0>(
                        fragment) + buffer->header_length;
158                 if(global_offset != expected_offset) {
159                     // There's a hole or overlap in the
                            data, bail out.
160                     buffer->reset();
161                     return nullptr;
162                 }
163                 std::size_t length = std::get<2>(fragment);
164                 if((global_offset + length) > newpkt_data.
                        length()) {
165                     // Nasty data, bail out.
166                     buffer->reset();
167                     return nullptr;
168                 }
169                 expected_offset += length;
170
171                 std::size_t local_offset = std::get<1>(
                        fragment);
172                 packet_ptr ptr = std::move(std::get<3>(
                        fragment));
173
174                 std::copy(ptr->data().begin() +
                        local_offset,
175                           ptr->data().begin() +
                                local_offset + length,
```

```
176                                 newpkt_data.begin() +
                                        global_offset);
177                     }
178                     auto final_length = buffer->total_length +
                            buffer->header_length;
179                     if(expected_offset != final_length) {
180                         // Data missing, bail out.
181                         buffer->reset();
182                         return nullptr;
183                     }
184
185                     // Note that at this point, the original header
                            will be wrong. We update the length but
186                     // no other fields, since the next processing
                            step might will most likely pull the header
187                     // off anyway.
188                     newpkt_data.set_bytes_be(+2, final_length);
189
190                     // Since the fragment was reassembled, we could
                            remove the buffer from the buffer map,
191                     // but this would require that we re-get the
                            top-level lock first, after dropping the
192                     // per-buffer lock. This would mean that we
                            need to block right here, possibly delaying
193                     // packet delivery further. Instead, we choose
                            simply reset the fragment buffer, reclaming
194                     // its resources. The resulting fragment buffer
                            will take up very little space, and will
195                     // be reclaimed next time the table is pruned.
196                     buffer->reset();
197
198                     // The resulting packet is now baked and ready
                            for returning
199                     return std::move(newpkt);
200                 }
201             }
202
203             return nullptr;
204         }
205
206         // Prunes the fragment buffers of old entries. Should be
                called periodically by the client software.
207         void prune_tables() {
208             std::unique_lock<decltype(frag_buffer_lock_)> guard(
                    frag_buffer_lock_);
209             // FUTURE: Make the cut-off timeout configurable
210             auto cutoff = fragment_clock::now() - std::chrono::
                    seconds(10);
211             // Note that because we are using gc ptrs, we can
                    remove them from the
212             // map without the underlying data going away, allowing
                    others to finish. We still need
213             // to lock when we check the timestamp however.
214             for(auto cur = frag_buffers_.cbegin(); cur !=
                    frag_buffers_.cend(); ) {
```

```
215                     std::unique_lock<decltype(fragment_buffer::lock)>
                            guard(cur->second->lock);
216                     if(cur->second->last_time < cutoff) {
217                         // Note that the gc pinters will make sure the
                                actual buffer sticks around for a bit
                                longer.
218                         frag_buffers_.erase(cur++);
219                     } else {
220                         ++cur;
221                     }
222                 }
223             }
224
225         private:
226
227             // OPTIMIZATION: Change this to a data structure with finer
                    grained locking. Seldom used path,
228             // so not a clear win.
229             std::mutex frag_buffer_lock_;
230             std::map<buffer_identifier, gc::gc_ptr<fragment_buffer>>
                    frag_buffers_;
231         };
232
233 } }
234
235 #endif
```

Listing C.22: trokis/protocols/ipv4/ipv4.hpp

```
1  //
2  //  ipv4.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ipv4_hpp
10 #define trokis_ipv4_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/packet.hpp"
14 #include "trokis/support/dispatcher.hpp"
15 #include "trokis/protocols/ipv4/types.hpp"
16 #include "trokis/protocols/ipv4/defragmenter.hpp"
17 #include "trokis/protocols/ipv4/upper/icmp4.hpp"
18
19 #include <unordered_map>
20 #include <algorithm>
21 #include <tuple>
22 #include <list>
23 #include <mutex>
24
25 namespace trokis { namespace ipv4 {
26
```

```
27      //
28      // Implements IPv4 functionality, including fragmentation and
            adressing. Currently does not
29      // perform any handling of options or multicast(but there is no
            filtering of output traffic,
30      // meaning that this is a type 1 multicast implementation. It
            is also possible to receive
31      // multicast traffic, but no multicast routing is done.).
32      //
33      // We choose not to implement options, as they are very
            seldomly used in moderne applications.
34      // While RFC791 specifies that options *must* be implemented by
            all nodes, we feel that a
35      // solution of simply forwarding the packets verbatim will
            comply to RFC1812 section 5.3.13:
36      //  "Unrecognized IP options in forwarded packets MUST be
            passed through unchanged."
37      // While RFC1812 also states several options that are mandatory
            , namely Source Route,
38      // Record Route and Timestamp. In all three cases, RFC1812 also
            specifies that the software
39      // may provide configuration to turn off the intrepretation of
            the option. Ignoring IP options
40      // thus won't cause any actual harm to the network
            transmissions. For these reasons, our
41      // implementation simply forwards options verbatim, without
            interpreting them at all. The result
42      // of this choice is a simpler implementation.
43      //
44      class ipv4_protocol : public utilities::generic_block<
45          packet_verdict(packet_ptr&&, protonum_t, address_t,
                address_t),
46          packet_verdict(packet_ptr&&, address_t)
47      > {
48      public:
49          ipv4_protocol()
50          : may_forward_(false) {}
51
52          packet_verdict icmp_reply(packet_ptr&& pkt, uint8_t type,
                uint8_t code,
53                  address_t src, address_t dst, int header_length) {
54              auto& pkt_data = pkt->data();
55              // Trim back the original packet header
56              pkt_data.trim(pkt_data.length() - header_length - 8);
57
58              // Push the ICMP checksum on
59              pkt_data.push(8);
60              pkt_data.set_byte(+0, type);
61              pkt_data.set_byte(+1, code);
62              pkt_data.set_bytes_be(+2, (uint16_t)0);
63
64              // Calculate the checksum
65              auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.begin(), pkt_data.end());
66              pkt_data.set_bytes_be(+2, checksum);
```

```
67
68              return transmit(std::move(pkt), src, dst, 1);
69          }
70
71          // Performs the local input part of the receive path
72          packet_verdict receive_local(packet_ptr&& pkt, std::size_t
                header_length,
73                                       address_t src, address_t dst)
                                           {
74              auto protocol = pkt->data().get_byte(+9);
75              pkt = defragmenter_.fragment_receive(std::move(pkt),
                    header_length, protocol, src, dst);
76              if(!pkt) {
77                  return packet_verdict::success; // Packet was
                        consumed by the defragmenter
78              }
79              // Might need to read out new header length
80              header_length = (pkt->data().get_byte(+0) & 0xF) * 4;
81              if(header_length < 20) {
82                  return packet_verdict::dropped;
83              }
84              pkt->data().pull(header_length);
85              return inform(std::move(pkt), protocol, src, dst);
86          }
87
88          // Performs the routing part of the receive path
89          packet_verdict receive_route(packet_ptr&& pkt, std::size_t
                header_length,
90                                       address_t src, address_t dst,
                                           dev_data* device_data) {
91              auto& pkt_data = pkt->data();
92
93              // FUTURE: Implement IPv4 options. Currently we simply
                    forward them verbatim.
94
95              // Decrement packet TTL
96              auto ttl = pkt_data.get_byte(+8);
97              if(--ttl == 0) {
98                  icmp_reply(std::move(pkt), 11, 0, src, dst,
                        header_length);
99                  return packet_verdict::dropped;
100             }
101             pkt_data.set_byte(+8, ttl);
102
103             auto route = get_routing_table().snapshot()->
                    find_route_lpm(dst);
104             if(route == nullptr) {
105                 icmp_reply(std::move(pkt), 3, 0, src, dst,
                        header_length);
106                 return packet_verdict::dropped;
107             }
108
109             auto nexthop = route->direct ? dst : route->
                    nexthop_addr;
110
```

```
111             pkt->dev = route->dev;
112             return output_lower(std::move(pkt), header_length,
                    nexthop, device_data);
113         }
114
115         // Recalculates the checksum of an IP packet. The packet is
                assumed to have already been length checked.
116         void recalc_checksum(packet_data& pkt_data, std::size_t
                header_length) {
117             // OPTIMIZATION: Change this to use incremental
                    updating of checksum, per RFC1624, or even use a
118             // stateless offload on the NIC.
119             pkt_data.set_bytes_be(+10, uint16_t(0));
120             auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.cbegin(), pkt_data.cbegin() +
                    header_length);
121             pkt_data.set_bytes_be(+10, uint16_t(checksum));
122         }
123
124         // If required by the given interface metrics, fragment an
                IPv4 packet and return both parts.
125         // If fragmentation is not required, only a single packet
                is returned. The first returned packet
126         // fragment will always be compliant to the given metrics(
                unless the interface cannot possibly
127         // accomodate IPv4), while the second fragment may be
                larger than the given interface MTU. The
128         // caller should repeatedly call fragment_packet with the
                second packet of the last call, until
129         // only a single packet is returned. Does NOT fix packet
                checksums!
130         std::pair<packet_ptr, packet_ptr> fragment_packet(
                packet_ptr&& pkt, std::size_t header_length,
131                                                             std::
                                                                tuple
                                                                <std
                                                                ::
                                                                size_t
                                                                , std
                                                                ::
                                                                size_t
                                                                >&
                                                                metrics
                                                                )
132         {
133             std::size_t maximum_payload;
134             std::size_t front_reserve;
135             std::tie(maximum_payload, front_reserve) = metrics;
136
137             auto retval = std::make_pair<packet_ptr, packet_ptr>(
                    nullptr, nullptr);
138             auto& pkt_data = pkt->data();
139             if(maximum_payload < pkt_data.length()) {
140                 // Can we fragment this packet?
141                 auto pkt1_flags = pkt_data.get_byte(+6);
```

```
142                 if(pkt1_flags & 0x40) { // Mask 0100 0000, bit 1
                        from MSB is "Don't Fragment"
143                     return retval; // Return if we are not allowed
                            to fragment
144                 }
145
146                 // Interface can't support IPv4
147                 if(maximum_payload < (header_length + 8)) {
148                     return retval;
149                 }
150
151                 // The number of bytes we can put in the payload
152                 auto payload_length = maximum_payload -
                        header_length;
153                 auto split_at = payload_length + header_length;
154                 auto nfb = payload_length / 8; // Number of
                        Fragment Blocks
155                 auto excess_length = pkt_data.length() - split_at;
156
157                 // Perform the fragmentation split
158                 auto pkt2 = std::make_unique<packet>();
159                 pkt2->dev = pkt->dev;
160                 pkt2->worker = pkt->worker;
161                 auto& pkt2_data = pkt2->data();
162                 pkt2_data.reserve_front(front_reserve);
163                 pkt2_data.put(header_length + excess_length);
164                 // Copy data
165                 std::copy_n(pkt_data.begin() + split_at,
                        excess_length, pkt2_data.begin() +
                        header_length);
166                 // Copy header
167                 std::copy_n(pkt_data.begin(), header_length,
                        pkt2_data.begin());
168                 pkt_data.trim(excess_length);
169
170                 // Fix up headers in the two new packets
171                 // Set More Fragments on the _first_ packet, don't
                        change it on the second.
172                 pkt1_flags |= 0x20; // Mask 0010 0000, bit 2 from
                        MSB is "More Fragments"
173                 pkt_data.set_byte(+6, pkt1_flags);
174
175                 // Set Total Length fields of packets
176                 pkt_data.set_bytes_be(+2, (uint16_t)split_at);
177                 pkt2_data.set_bytes_be(+2, (uint16_t)(header_length
                        + excess_length));
178
179                 // Set Fragment Offset in _second_ packet to the
                        outer packet offset + nfb, don't change first.
180                 auto fragment_offset = pkt2_data.get_bytes_be
                        <2>(+6);
181                 // Preserve top three flag bits.
182                 fragment_offset = (0xE000 & fragment_offset) | (0
                        x1FFF & ((fragment_offset & (0x1FFF)) + nfb));
```

```
183                 pkt2_data.set_bytes_be(+6, (uint16_t)
                        fragment_offset);
184
185                 retval.second = std::move(pkt2);
186             }
187             retval.first = std::move(pkt);
188             return retval;
189         }
190
191         // Returns a tuple with element 0 indicating the maximum
                payload that should be sent and
192         // element 1 indicating the amount of room that should be
                reserved in front of the IP header.
193         std::tuple<std::size_t, std::size_t> get_packet_metrics(
                address_t nexthop, device_id dev)
194         {
195             // FUTURE: Make this use the neighbor infrastructure to
                    override interface metrics
196             return std::make_tuple(1500, 14);
197         }
198
199         packet_verdict output_lower_one(packet_ptr&& pkt, std::
                size_t header_length, address_t nexthop) {
200             recalc_checksum(pkt->data(), header_length); //
                    Recalculate checksum
201             return request(std::move(pkt), nexthop);      // Output
202         }
203
204         // Performs delivery to the lower layer. In case the MTU of
                 the device is too low, this can fail
205         // and this performs the required fragmentation.
206         packet_verdict output_lower(packet_ptr&& pkt, std::size_t
                header_length, address_t nexthop, dev_data* ddata) {
207             auto metrics = get_packet_metrics(nexthop, pkt->dev);
208             if(pkt->data().length() <= std::get<0>(metrics)) { //
                    Fast path: no fragmentation
209                 return output_lower_one(std::move(pkt),
                        header_length, nexthop);
210             }
211             auto pkts = fragment_packet(std::move(pkt),
                    header_length, metrics);
212             for(;;) {
213                 if(!pkts.first) {
214                     // Fragmentation failed because of a Dont
                            Fragment bit, so send ICMP to the sender
215                     // if not local. Currently we don't check for
                            this being to a local address.
216                     auto dst = pkt->data().get_bytes<4>(+12);
217                     auto src = ddata->get_primary_address();
218                     icmp_reply(std::move(pkt), 3, 4, src, dst,
                            header_length);
219                     return packet_verdict::dropped;
220                 }
221                 auto verdict = output_lower_one(std::move(pkts.
                        first), header_length, nexthop);
```

```
222                 if(verdict != packet_verdict::success) {
223                     // A packet failed on output, return verdict
                            and stop processing.
224                     return verdict;
225                 }
226
227                 // Are there more packets to process?
228                 if(pkts.second) {
229                     pkts = fragment_packet(std::move(pkts.second),
                            header_length, metrics);
230                 } else {
231                     // If not, break the loop
232                     break;
233                 }
234             }
235             return packet_verdict::success;
236         }
237
238         template<typename... Args>
239         dev_data* get_or_create_dev_data(device_id id, Args&&...
                args) {
240             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
241             auto retval = dev_data_.emplace(id, gc::make_gc_ptr<
                    dev_data>(std::forward<Args>(args)...));
242             return retval.first->second.acquire();
243         }
244
245         void drop_dev_data(device_id id) {
246             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
247             // Note that the actual dev_data entry will be
                    reclaimed later by gc_ptr
248             dev_data_.erase(id);
249         }
250
251         dev_data* get_device_data(device_id id) {
252             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
253             auto iter = dev_data_.find(id);
254             if(iter == dev_data_.end()) return nullptr;
255             return iter->second.acquire();
256         }
257
258         // IPv4 input path. Performs basic safety checks on the
                packet and forwards it to either
259         // the routing or the local input path. If the packet is in
                malformed, it will be dropped here.
260         packet_verdict receive(packet_ptr&& pkt, address_type
                linklayer_type) {
261             auto* device_data = get_device_data(pkt->dev);
262             if(device_data == nullptr) return packet_verdict::
                    dropped;
263
264             /* Parse IPv4 packet per RFC 791 sec. 3.1 */
```

```
265             auto& pkt_data = pkt->data();
266
267             // Minimum size of IPv4 header is 4*5 = 20 bytes, so
                    check that we may
268             // pull at least that. (RFC1812 sec 5.2.2)
269             if(!pkt_data.may_pull(20)) {
270                 return packet_verdict::dropped;
271             }
272
273             // First byte: 4 bits of version followed by 4 bits of
                    header length(RFC791 sec 3.1)
274             auto first_byte = pkt_data.get_byte(+0);
275             if((first_byte & 0xF0) != 0x40) {
276                 // Not an IPv4 packet
277                 return packet_verdict::dropped;
278             }
279
280             // Verify that packet header length is OK (RFC1812 sec
                    5.2.2)
281             uint8_t header_length = (first_byte & 0xF) * 4;
282             if(header_length < 20) {
283                 return packet_verdict::dropped;
284             }
285
286             // Verify packet length (RFC1812 sec 5.2.2)
287             uint16_t total_length = pkt_data.get_bytes_be<2>(+2);
288             if(total_length < header_length) {
289                 return packet_verdict::dropped;
290             }
291             if(!pkt_data.may_pull(total_length)) {
292                 return packet_verdict::dropped;
293             }
294             // Trim back end of packet. Note that we already
                    ensured that the packet was atleast total_length.
295             auto excess_length = pkt_data.length() - total_length;
296             if(excess_length > 0) {
297                 pkt_data.trim(excess_length);
298             }
299
300             // Verify checksum, must silently drop failing packets
                    (RFC1812 sec 4.2.2.5)
301             auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.cbegin(), pkt_data.cbegin() +
                    header_length);
302             if(checksum != 0) {
303                 return packet_verdict::dropped;
304             }
305
306             // Read addresses
307             auto src = pkt_data.get_bytes<4>(+12);
308             auto dst = pkt_data.get_bytes<4>(+16);
309
310             // Note: we do _not_ check TTL here, RFC1812 sec.
                    4.2.2.9 specifies that we must not inspect TTL
                    unless
```

```
311             // we are routing.
312             if(!device_data->is_local_address(dst) && dst !=
                    addr_global_broadcast) {
313                 // Perform routing. We don't modify the header if
                        we can avoid it, so don't pull from the packet.
314                 if(!may_forward_) return packet_verdict::dropped;
315                 return receive_route(std::move(pkt), header_length,
                        src, dst, device_data);
316             } else {
317                 return receive_local(std::move(pkt), header_length,
                        src, dst);
318             }
319         }
320
321         // Transmit path for upper level
322         packet_verdict transmit(packet_ptr&& pkt, address_t src,
                address_t dst, protonum_t proto) {
323             auto& data = pkt->data();
324             if(data.length() > (65535 - 20)) { // Maximum length of
                    payload is 2^16-1 - header length
325                 return packet_verdict::dropped;
326             }
327
328             auto route = get_routing_table().snapshot()->
                    find_route_lpm(dst);
329             if(route == nullptr) { // Can't find a matching route
330                 return packet_verdict::dropped;
331             }
332             auto* device_data = get_device_data(route->dev);
333             if(device_data == nullptr) return packet_verdict::
                    dropped;
334
335
336             // Append 20 bytes of IPv4 header and fill it in
337             data.push(20);
338             data.set_byte(+0, 0x45); // IP version 4 and a header
                    length of 20 bytes. Options are not supported.
339             data.set_byte(+1, 0x00); // DSCP and ECN are set to
                    zero
340             data.set_bytes_be(+2, (uint16_t)data.length());
341             data.set_bytes_be(+4, (uint16_t)(rand() & 0xFFFF)); //
                    Set the identification field to a random value
342             data.set_bytes_be(+6, (uint16_t)0); // FUTURE: Add a
                    way to set Dont Fragment here, for PMTU Discovery
343             data.set_byte(+8, 64); // Set TTL
344             data.set_byte(+9, (uint8_t)proto);
345             data.set_bytes(+12, src);
346             data.set_bytes(+16, dst);
347
348             auto nexthop = route->direct ? dst : route->
                    nexthop_addr;
349             pkt->dev = route->dev;
350             return output_lower(std::move(pkt), 20, nexthop,
                    device_data);
351         }
```

```
352
353         routing_table& get_routing_table() {
354             return routing_table_;
355         }
356
357         // Helper for external systems to get the primary address
                of a device. Mainly for ARP.
358         address_t get_primary_address(device_id dev) {
359             auto dev_data = get_device_data(dev);
360             if(dev_data == nullptr) return address_t();
361             return dev_data->get_primary_address();
362         }
363
364         bool is_local_address(device_id dev, address_t addr) {
365             auto dev_data = get_device_data(dev);
366             if(dev_data == nullptr) return false;
367             return dev_data->is_local_address(addr);
368         }
369
370         // Enable or disable gateway functionality
371         void set_gateway(bool flag) {
372             may_forward_ = flag;
373         }
374     private:
375         std::atomic<bool> may_forward_;
376
377         routing_table routing_table_;
378         defragmenter defragmenter_;
379
380         std::mutex dev_data_lock_;
381         std::unordered_map<device_id, gc::gc_ptr<dev_data>>
                dev_data_;
382     };
383
384 } }
385
386 #endif
```

Listing C.23: trokis/protocols/ipv4/types.hpp

```
1  //
2  //  ipv4_types.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 12/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ipv4_types_hpp
10 #define trokis_ipv4_types_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/support/gc.hpp"
14 #include "trokis/support/routing_table.hpp"
15
```

```
16 #include <array>
17 #include <iterator>
18 #include <algorithm>
19 #include <cassert>
20
21 namespace trokis { namespace ipv4 {
22
23     using address_t = std::array<byte_t, 4>;
24     using protonum_t = uint8_t;
25
26     static constexpr address_t addr_global_broadcast {{255, 255,
           255, 255}};
27
28     class routing_table {
29     public:
30         routing_table()
31         : routing_table_(gc::make_gc_ptr<decltype(routing_table_)::
               element_type>()) { }
32
33         routing::routing_table<address_t>* snapshot() const {
34             return routing_table_.acquire();
35         }
36
37         void add_route(const routing::route<address_t>& route) {
38             routing_table_.update([&](decltype(routing_table_)::
                   element_type& table) {
39                 table.add_route(route);
40             });
41         }
42
43     private:
44         gc::gc_ptr<routing::routing_table<address_t>>
               routing_table_;
45     };
46
47     class dev_data {
48     public:
49         dev_data() : unicast_addresses_(gc::make_gc_ptr<decltype(
               unicast_addresses_)::element_type>())
50         { }
51
52         bool is_local_address(address_t addr) {
53             auto& current_list = *unicast_addresses_;
54             auto retval = std::find(std::begin(current_list), std::
                   end(current_list), addr);
55             return retval != std::end(current_list);
56         }
57
58         void local_address_add(address_t addr) {
59             unicast_addresses_.update([&](decltype(
                   unicast_addresses_)::element_type& list) {
60                 list.push_back(addr);
61             });
62         }
63
```

```
64          address_t get_primary_address() {
65              auto& current_list = *unicast_addresses_;
66              return current_list.size() != 0 ? *current_list.begin()
                    : address_t();
67          }
68
69      private:
70          gc::gc_ptr<std::vector<address_t>> unicast_addresses_;
71      };
72
73  } }
74
75  #endif
```

Listing C.24: trokis/protocols/ipv4/upper/icmp4.hpp

```
1   //
2   //  icmp4.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 17/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_icmp4_hpp
10  #define trokis_icmp4_hpp
11
12  namespace trokis { namespace ipv4 { namespace icmp4 {
13
14      // Implements basic parts of RFC792
15      class icmp4_protocol
16      : public utilities::generic_block_request_hook<
17        packet_verdict(packet_ptr&& pkt, address_t dst, address_t src
              , protonum_t proto)
18      > {
19      public:
20
21          // Receives an ICMPv4 packet and performs required parsing.
22          packet_verdict receive(packet_ptr&& pkt, address_t src,
                address_t dst) {
23              auto& pkt_data = pkt->data();
24              if(!pkt_data.may_pull(28)) { // Length of ICMP header +
                     shortest IP header
25                  return packet_verdict::dropped;
26              }
27
28              auto type = pkt_data.get_byte(+0);
29              auto code = pkt_data.get_byte(+1);
30              // auto checksum = pkt_data.get_bytes_be<2>(+2);
31              pkt_data.pull(4);
32              switch(type) {
33              case 8:
34                  if(code != 0) return packet_verdict::dropped;
35                  return receive_ping(std::move(pkt), src, dst);
36                  break;
```

```
37              default:
38                  return packet_verdict::dropped;
39                  break;
40              }
41          }
42
43      private:
44          void encap_icmp(packet_data& pkt_data, uint8_t type,
                uint8_t code) {
45              pkt_data.push(4);
46              pkt_data.set_byte(+0, type);
47              pkt_data.set_byte(+1, code);
48          }
49
50          void fix_checksum(packet_data& pkt_data) {
51              pkt_data.set_bytes_be(+2, (uint16_t)0);
52              auto checksum = byte_tools::checksum_ones_complement(
                    pkt_data.begin(), pkt_data.end());
53              pkt_data.set_bytes_be(+2, (uint16_t)checksum);
54          }
55
56          packet_verdict receive_ping(packet_ptr&& pkt, address_t src
                , address_t dst) {
57              // In this case we modify the original packet and send
                    it back out.
58              encap_icmp(pkt->data(), 0, 0);
59              fix_checksum(pkt->data());
60              return request(std::move(pkt), dst, src, 1);
61          }
62      };
63
64
65  } } }
66
67  #endif
```

Listing C.25: trokis/protocols/ipv6/ipv6.hpp

```
1   //
2   //  ipv6.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 09/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_ipv6_hpp
10  #define trokis_ipv6_hpp
11
12  #include "trokis/types.hpp"
13  #include "trokis/protocols/ipv6/types.hpp"
14
15  namespace trokis { namespace ipv6 {
16
17      //
```

```
18      // Implements the IP Layer of the IPv6 protocol(RFC 2460) as
            required by RFC 4294.
19      //
20      // Our implementation ignores the Flow Label field as specified
            by RFC 2460, forwarding
21      // it unchanged, originating packets with it set to zero and
            ignoring it on reception,
22      // as allowed by RFC 2460 section 6.
23      //
24      // Our implementation reads the traffic class field and
            forwards it to the upper layer
25      // protocols, but it does not make any attempt to decode the
            meaning of the field. We
26      // also allow upper layer protocols to supply a value to the
            field.
27      //
28      // Unlike the IPv4 implementation, we do not implement any IPv6
            options here. Instead,
29      // we let upper level handlers manage these. This is made
            possible by the fact that IPv6
30      // does not use an embedded list of options but instead uses a
            linked list of headers
31      // with every header containing the type of the next header in
            the list. When forwarding
32      // is implemented, the Hop-by-Hop option should be managed in
            in this module.
33      //
34      class ipv6_protocol : utilities::generic_block<
35          packet_verdict(packet_ptr&&, protonum_t, address_t,
                address_t, byte_t),
36          packet_verdict(packet_ptr&&, address_t)
37      >{
38      public:
39          packet_verdict receive_local(packet_ptr&& pkt, address_t
                dst) {
40              auto& pkt_data = pkt->data();
41
42              // Perform options parsing
43              auto next_header = pkt_data.get_byte(+6);
44              if(next_header == 59) return packet_verdict::success;
45
46              // Get the data we need to pass to upper layer
47              auto src = pkt_data.get_bytes<16>(+8);
48              auto tc = (pkt_data.get_bytes_be<2>(+0) >> 4) & 0xFF;
49              pkt_data.pull(40); // Pull off the IPv6 header
50
51              return inform(std::move(pkt), next_header, src, dst, tc
                    );
52          }
53
54          packet_verdict receive(packet_ptr&& pkt, address_type
                linklayer_type) {
55              auto dev_data = get_device_data(pkt->dev);
56              if(dev_data == nullptr) {
57                  return packet_verdict::dropped;
```

```
58              }
59
60              auto& pkt_data = pkt->data();
61
62              // Verify that the packet can fit at least the IPv6
                    header
63              if(!pkt_data.may_pull(40)) {
64                  return packet_verdict::dropped;
65              }
66
67              // Read version and verify it
68              if((pkt_data.get_byte(+0) >> 4) != 6) {
69                  return packet_verdict::dropped; // Not IPv6
70              }
71
72              // Get payload length
73              auto payload_length = pkt_data.get_bytes_be<2>(+4);
74              auto total_length = payload_length + 40;
75              if(!pkt_data.may_pull(total_length)) {
76                  return packet_verdict::dropped;
77              }
78              // Trim packet to correct length
79              pkt_data.trim(pkt_data.length() - total_length);
80
81              // Get destination address
82              auto dst = pkt_data.get_bytes<16>(+24);
83
84              if(dev_data->is_local_address(dst)) {
85                  return receive_local(std::move(pkt), dst);
86              } else {
87                  return packet_verdict::dropped;
88              }
89          }
90
91          packet_verdict transmit(packet_ptr&& pkt, address_t dst,
                address_t src,
92                  byte_t traffic_class, protonum_t next_header) {
93              auto& pkt_data = pkt->data();
94
95              auto length = pkt_data.length();
96              if(length > 0xFFFF) {
97                  return packet_verdict::dropped;
98              }
99
100             auto route = get_routing_table().snapshot()->
                    find_route_lpm(dst);
101             if(route == nullptr) { // Can't find a matching route
102                 return packet_verdict::dropped;
103             }
104             auto dev_data = get_device_data(route->dev);
105             if(dev_data == nullptr) {
106                 return packet_verdict::dropped;
107             }
108
109             // Insert IPv6 header
```

```
110             pkt_data.push(40);
111             // Version, traffic class and flow label(= 0)
112             uint32_t first_word = (4 << 28) | (traffic_class << 20)
                    ;
113             pkt_data.set_bytes_be(+0, first_word);
114
115             // Payload length
116             pkt_data.set_bytes_be(+4, (uint16_t)length);
117             pkt_data.set_byte(+6, next_header);
118             pkt_data.set_byte(+7, 64);
119             pkt_data.set_bytes(+8, src);
120             pkt_data.set_bytes(+24, dst);
121
122             auto nexthop = route->direct ? dst : route->
                    nexthop_addr;
123             pkt->dev = route->dev;
124             return request(std::move(pkt), nexthop);
125         }
126
127         routing_table& get_routing_table() {
128             return routing_table_;
129         }
130
131         template<typename... Args>
132         dev_data* get_or_create_dev_data(device_id id, Args&&...
                args) {
133             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
134             auto retval = dev_data_.emplace(id, gc::make_gc_ptr<
                    dev_data>(std::forward<Args>(args)...));
135             return retval.first->second.acquire();
136         }
137
138         void drop_dev_data(device_id id) {
139             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
140             // Note that the actual dev_data entry will be
                    reclaimed later by gc_ptr
141             dev_data_.erase(id);
142         }
143
144         dev_data* get_device_data(device_id id) {
145             std::unique_lock<decltype(dev_data_lock_)> guard(
                    dev_data_lock_);
146             auto iter = dev_data_.find(id);
147             if(iter == dev_data_.end()) return nullptr;
148             return iter->second.acquire();
149         }
150
151     private:
152         routing_table routing_table_;
153
154         std::mutex dev_data_lock_;
155         std::unordered_map<device_id, gc::gc_ptr<dev_data>>
                dev_data_;
```

```
156     };
157
158 } }
159
160 #endif
```

Listing C.26: trokis/protocols/ipv6/types.hpp

```
1  //
2  //  ipv6_types.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 12/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_ipv6_types_hpp
10 #define trokis_ipv6_types_hpp
11
12 #include "trokis/types.hpp"
13 #include "trokis/support/routing_table.hpp"
14
15 namespace trokis { namespace ipv6 {
16
17     using address_t = std::array<byte_t, 16>;
18     using protonum_t = byte_t;
19     using scope_t = byte_t;
20
21     class routing_table {
22     public:
23         routing_table()
24         : routing_table_(gc::make_gc_ptr<decltype(routing_table_)::
              element_type>()) { }
25
26         routing::routing_table<address_t>* snapshot() const {
27             return routing_table_.acquire();
28         }
29
30         void add_route(const routing::route<address_t>& route) {
31             routing_table_.update([&](decltype(routing_table_)::
                  element_type& table) {
32                 table.add_route(route);
33             });
34         }
35
36     private:
37         gc::gc_ptr<routing::routing_table<address_t>>
              routing_table_;
38     };
39
40     // Fat POD type for containing data describing an interface.
          Helpers are provided for all data
41     // that needs locking.
42     class dev_data {
43     public:
```

```
44          using address_desc_type = std::tuple<address_t,scope_t>;
45
46          dev_data() : local_addresses_(gc::make_gc_ptr<decltype(
                local_addresses_)::element_type>())
47          { }
48
49          bool is_local_address(address_t addr) {
50              auto& current_list = *local_addresses_;
51              auto retval = std::find_if(std::begin(current_list),
                    std::end(current_list),
52                  [&] (const std::tuple<address_t,scope_t>& val) {
                        return std::get<0>(val) == addr; });
53              return retval != std::end(current_list);
54          }
55
56          void local_address_add(address_t addr, scope_t scope) {
57              local_addresses_.update([&](decltype(local_addresses_)
                    ::element_type& list) {
58                  list.emplace_back(addr, scope);
59              });
60          }
61
62          void local_addresses_remove(address_t addr) {
63              local_addresses_.update([&](decltype(local_addresses_)
                    ::element_type& list) {
64                  std::remove_if(std::begin(list), std::end(list),
65                      [&] (const std::tuple<address_t,scope_t>& val)
                            { return std::get<0>(val) == addr; });
66              });
67          }
68
69          std::vector<address_desc_type> get_local_addresses() {
70              auto& current_list = *local_addresses_;
71              return std::vector<address_desc_type>(std::begin(
                    current_list), std::end(current_list));
72          }
73
74          void set_iface_identifier(std::array<byte_t, 8> id) {
75              iface_identifier_ = id;
76          }
77
78          std::array<byte_t, 8> iface_identifier() {
79              return iface_identifier_;
80          }
81
82      private:
83          gc::gc_ptr<std::vector<address_desc_type>> local_addresses_
                ;
84          std::array<byte_t, 8> iface_identifier_;
85      };
86
87  } }
88
89  #endif
```

Listing C.27: trokis/support/byte_swappers.hpp

```
//
//  byte_swappers.hpp
//  trokis
//
//  Created by Lasse Bang Dalegaard on 12/07/13.
//  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
//

#ifndef trokis_byte_swappers_hpp
#define trokis_byte_swappers_hpp

#include "trokis/types.hpp"

#include <algorithm>

namespace trokis { namespace byte_tools {

    // Generic byte-swapping mechanism based on std::reverse. Reverses
    // a byte array, in effect performing a byte swap.
    template<std::size_t N>
    struct byte_swapper_reverse {
        inline static void swap(std::array<byte_t, N>& raw) {
            std::reverse(raw.begin(), raw.begin() + N);
        }
    };

} }

#endif
```

Listing C.28: trokis/support/byte_tools.hpp

```
//
//  byte_tools.hpp
//  trokis
//
//  Created by Lasse Bang Dalegaard on 09/07/13.
//  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
//

#ifndef trokis_byte_tools_hpp
#define trokis_byte_tools_hpp

#include <boost/detail/endian.hpp>
#include "trokis/support/byte_swappers.hpp"

#include <algorithm>

namespace trokis { namespace byte_tools {

    struct endianness_little;
    struct endianness_big;

```

```
22 #if defined(BOOST_LITTLE_ENDIAN)
23     using endianness_host = endianness_little;
24 #elif defined(BOOST_BIG_ENDIAN)
25     using endianness_host = endianness_big;
26 #else
27 #error "System endianness unknown"
28 #endif
29
30     // Identity byte swapper(ie. does nothing)
31     template<std::size_t N>
32     struct byte_swapper_id {
33         static void swap(std::array<byte_t, N>& raw) { }
34     };
35
36     // Generic byte swapping interface
37     template<std::size_t N, typename SourceEndianness, typename
           DestinationEndianness>
38     struct byte_swapper;
39
40     // Byte swapping specializations to/from little/endian. Uses
           reversing or identity.
41     template<std::size_t N>
42     struct byte_swapper<N, endianness_little, endianness_little> :
           byte_swapper_id<N> { };
43     template<std::size_t N>
44     struct byte_swapper<N, endianness_little, endianness_big> :
           byte_swapper_reverse<N> { };
45     template<std::size_t N>
46     struct byte_swapper<N, endianness_big, endianness_little> :
           byte_swapper_reverse<N> { };
47     template<std::size_t N>
48     struct byte_swapper<N, endianness_big, endianness_big> :
           byte_swapper_id<N> { };
49
50     template<std::size_t N>
51     struct nbyte_type {
52         using type = uint64_t;
53     };
54
55     template<>
56     struct nbyte_type<2> {
57         using type = uint16_t;
58     };
59
60     template<std::size_t N>
61     using nbyte_type_t = typename nbyte_type<N>::type;
62
63     // Directly reads bytes, no swapping of endianness or similar
           is performed
64     template<std::size_t N, typename ForwardIt>
65     std::array<uint8_t, N> read_bytes_array(const ForwardIt& begin)
           {
66         std::array<byte_t, N> tmp;
67         std::copy(begin, begin + N, tmp.begin());
68         return tmp;
```

```
69      }
70
71      // Generic read_bytes call taking the forward input iterator
            for a container. Does _not_ perform bounds checking.
72      template<std::size_t N, typename Endianness, typename ForwardIt
            >
73      nbyte_type_t<N> read_bytes(ForwardIt&& begin) {
74          auto tmp = read_bytes_array<N>(std::forward<ForwardIt>(
                begin));
75          asm("#preswap");
76          byte_swapper<N, Endianness, endianness_host>::swap(tmp);
77          asm("#postswap");
78          return *((nbyte_type_t<N>*)tmp.data());
79      }
80
81      // Directly writes bytes, no swapping of endianness or similar
            is performed
82      template<std::size_t N, typename ForwardIt>
83      void write_bytes_array(const std::array<byte_t, N>& data, const
             ForwardIt& begin) {
84          std::copy(data.begin(), data.end(), begin);
85      }
86
87
88      template<typename T>
89      struct convert_to_bytes_impl {
90          static constexpr std::size_t length = sizeof(T);
91
92          static std::array<byte_t, length> convert(T val) {
93              std::array<byte_t, length> tmp;
94              byte_t* raw = reinterpret_cast<byte_t*>(&val);
95              std::copy(raw, raw + length, tmp.begin());
96              return tmp;
97          }
98      };
99
100     // Generic write_bytes calls. Takes data and forward output
            iterator
101     template<typename Endianness, typename T, typename ForwardIt>
102     void write_bytes(T&& data, ForwardIt&& begin) {
103         using converter = convert_to_bytes_impl<T>;
104         auto raw = converter::convert(std::forward<T>(data));
105         byte_swapper<converter::length, endianness_host, Endianness
                >::swap(raw);
106         write_bytes_array(raw, std::forward<ForwardIt>(begin));
107     }
108
109     // Performs a ones-complement sum of all 2-octet words in the
            given iterator range. Used by IP, UDP and TCP
110     // for checksum calculations. See RFC 1071.
111     template<typename ForwardIt>
112     uint16_t checksum_ones_complement(const ForwardIt& begin, const
             ForwardIt& end) {
113         uint32_t csum = 0;
114         bool shift = true;
```

```
115         for(auto cur = begin; cur != end; ++cur) {
116             csum += *cur << (shift ? 8 : 0);
117             shift = !shift;
118         }
119         while(csum > 0xFFFF) {
120             csum = (csum & 0xFFFF) + (csum >> 16);
121         }
122         return ~(csum) & 0xFFFF;
123     }
124
125 } }
126
127 #endif
```

Listing C.29: trokis/support/dispatcher.hpp

```
1  //
2  //  dispatcher_func.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 09/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_dispatcher_func_hpp
10 #define trokis_dispatcher_func_hpp
11
12 #include <map>
13 #include <utility>
14 #include <functional>
15
16 namespace trokis { namespace utilities {
17
18     // Dispatcher functor. Strips out second argument of a function
           call and uses it
19     // as the index into a loopup table/map to determine the actual
           function to call.
20     template<typename>
21     class dispatcher;
22
23     template<typename SwitchType, typename RetType, typename...
           ArgTypes>
24     class dispatcher<RetType(packet_ptr&&, SwitchType, ArgTypes...)
           > {
25     public:
26         dispatcher() { }
27         dispatcher(const dispatcher&) = delete;
28
29         using result_type = RetType;
30         using func_type = RetType(packet_ptr&&, ArgTypes...);
31         using fallback_func_type = RetType(packet_ptr&&, SwitchType
               , ArgTypes...);
32         using wrapped_func_type = std::function<func_type>;
33         using fallback_wrapped_func_type = std::function<
               fallback_func_type>;
```

```
34          using map_type = std::map<SwitchType, wrapped_func_type>;
35
36          result_type operator()(packet_ptr&& p, SwitchType key,
                ArgTypes&&... args) {
37              auto func = find_handler(key);
38              if(func != funcs_.end()) {
39                  return std::get<1>(*func)(std::move(p), std::
                        forward<ArgTypes>(args)...);
40              } else {
41                  return fallback_(std::move(p), key, std::forward<
                        ArgTypes>(args)...);
42              }
43          }
44
45          template<typename Func>
46          void set_handler(SwitchType key, Func&& func) {
47              funcs_[key] = std::forward<Func>(func);
48          }
49
50          template<typename Func>
51          void set_fallback(Func&& func) {
52              fallback_ = std::forward<Func>(func);
53          }
54
55      private:
56          typename map_type::iterator find_handler(SwitchType key) {
57              return funcs_.find(key);
58          }
59
60          map_type funcs_;
61          fallback_wrapped_func_type fallback_;
62      };
63
64      // Default value for a hook with the given return type.
65      template<typename>
66      struct default_hook_return;
67
68      template<>
69      struct default_hook_return<packet_verdict> {
70          static constexpr auto value = packet_verdict::dropped;
71      };
72
73      // Default hook implementation, as a fallback when no key
            matched in a dispatcher.
74      template<typename RetType, typename... Args>
75      struct default_hook_impl;
76
77      template<typename RetType, typename... Args>
78      struct default_hook_impl<RetType(Args...)> {
79          RetType operator()(Args...) {
80              return default_hook_return<RetType>::value;
81          }
82      };
83
84      // Returns the default hook for a given function prototype.
```

```
85      template<typename Hook>
86      std::function<Hook> default_hook() {
87          return default_hook_impl<Hook>();
88      }
89
90      // Mixin for a generic request hook.
91      template<typename Request>
92      class generic_block_request_hook {
93      private:
94          using hook_request_t = std::function<Request>;
95      public:
96          generic_block_request_hook() :
97          hook_request_(default_hook<Request>())
98          { }
99
100         template<typename Func>
101         void set_hook_request(Func&& hook) {
102             hook_request_ = std::forward<Func>(hook);
103         }
104
105     protected:
106
107         template<typename... Args>
108         typename hook_request_t::result_type request(Args&&... args
                ) {
109             return hook_request_(std::forward<Args>(args)...);
110         }
111
112     private:
113         hook_request_t hook_request_;
114     };
115
116     // Mixin for a generic inform hook.
117     template<typename Inform>
118     class generic_block_inform_hook {
119     private:
120         using hook_inform_t = std::function<Inform>;
121     public:
122         generic_block_inform_hook() :
123         hook_inform_(default_hook<Inform>())
124         { }
125
126         template<typename Func>
127         void set_hook_inform(Func&& hook) {
128             hook_inform_ = std::forward<Func>(hook);
129         }
130
131     protected:
132
133         template<typename... Args>
134         typename hook_inform_t::result_type inform(Args&&... args)
                {
135             return hook_inform_(std::forward<Args>(args)...);
136         }
137
```

```
138     private:
139         hook_inform_t hook_inform_;
140     };
141 
142     // Mixin combining an Inform hook and a Request hook as a
            single mixin.
143     template<
144         typename Inform,
145         typename Request
146     >
147     class generic_block : public generic_block_inform_hook<Inform>,
148                           public generic_block_request_hook<Request
                                  >
149     { };
150 
151 } }
152 
153 #endif
```

Listing C.30: trokis/support/gc.hpp

```
1  //
2  //  gc.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 11/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8  
9  #ifndef trokis_gc_hpp
10 #define trokis_gc_hpp
11 
12 #include <memory>
13 #include <atomic>
14 #include <chrono>
15 #include <thread>
16 
17 #include "trokis/containers/intrusive_queue_mpsc.hpp"
18 #include "trokis/support/gc_reclaimer.hpp"
19 
20 namespace trokis { namespace gc {
21 
22     template<typename T>
23     class gc_ptr
24     {
25     public:
26         using element_type = T;
27 
28         // Constructs the pointer with a null value
29         gc_ptr() : ptr_(nullptr) { }
30 
31         // Constructs the pointer with a given value
32         gc_ptr(element_type* ptr) : ptr_(ptr) { }
33 
34         ~gc_ptr() {
```

```
35              if(ptr_ != nullptr) {
36                  reclaim(ptr_, static_default_delete<element_type>::
                        call);
37              }
38          }
39
40          // Move constructor
41          gc_ptr(gc_ptr<element_type>&& other) : ptr_(nullptr) {
42              assign(other.ptr_.load());
43              other.ptr_ = nullptr;
44          }
45
46          // Move assignment.
47          gc_ptr<element_type>& operator=(gc_ptr<element_type>&&
                other) {
48              assign(other.ptr_.load());
49              other.ptr_ = nullptr;
50              return *this;
51          }
52
53          // Acquires the pointer by performing a load with the
                proper memory barriers.
54          element_type* acquire() const {
55              return ptr_.load();
56          }
57
58          // Copies the pointed to object and performs the update in
                the supplied function object.
59          // Then tries to perform a CAS on the pointer value, and if
                 successful, returns. If the
60          // object was updated by another updater meanwhile, the
                function is restarted. In effect,
61          // a very simple form of software transactional memory is
                implemented.
62          //
63          // The supplied function object should NOT cause side
                effects, as it may be run multiple
64          // times.
65          //
66          // It is up to clients of this function to perform any
                locking that may be required to
67          // synchronize writers.
68          template<typename Func>
69          void update(Func&& f) {
70              while(1) {
71                  auto current = acquire();
72                  std::unique_ptr<element_type> updated;
73                  if(current) {
74                      updated = std::make_unique<element_type>(*
                            current);
75                  } else {
76                      updated = std::make_unique<element_type>();
77                  }
78                  f(*updated);
79                  if(compare_assign(current, updated.get())) {
```

```
80                      updated.release();
81                      return;
82                  }
83              }
84          }
85
86          // Performs an atomic CAS of the pointer, with the default
                deleter for the old object.
87          bool compare_assign(element_type* expected, element_type*
                ptr) {
88              return compare_assign(expected, ptr,
                    static_default_delete<element_type>::call);
89          }
90
91          // Performs an atomic CAS of the pointer, using a given
                deleter for the old object.
92          template<typename Deleter>
93          bool compare_assign(element_type* expected, element_type*
                ptr, Deleter&& deleter) {
94              auto success = ptr_.compare_exchange_weak(expected, ptr
                    );
95              if(success) {
96                  reclaim(expected, std::forward<Deleter>(deleter));
97                  return true;
98              } else {
99                  return false;
100             }
101         }
102
103         // Performs an atomic assignment of the pointer, using the
                default delter for the old object.
104         void assign(element_type* ptr) {
105             assign(ptr, static_default_delete<element_type>::call);
106         }
107
108         // Performs an atomic assignment of the pointer, using the
                given deleter for the old object.
109         template<typename Deleter>
110         void assign(element_type* ptr, Deleter&& deleter) {
111             auto oldptr = ptr_.exchange(ptr);
112             if(oldptr == nullptr) {
113                 return;
114             }
115             reclaim(oldptr, std::forward<Deleter>(deleter));
116         }
117
118         // See assign(element_type*)
119         gc_ptr<element_type>& operator=(element_type* ptr) {
120             assign(ptr);
121             return *this;
122         }
123
124         // See acquire()
125         element_type& operator*() const {
126             return *acquire();
```

```
127         }
128
129         // See acquire()
130         element_type* operator->() const {
131             return acquire();
132         }
133
134     private:
135         template<typename Deleter>
136         void reclaim(element_type* ptr, Deleter&& deleter) {
137             __global_reclaimer.reclaim(ptr, std::forward<Deleter>(
                    deleter));
138         }
139
140         std::atomic<element_type*> ptr_;
141     };
142
143     // std::make_shared-like constructor for gc_ptr. Forwards all
            arguments directly to
144     // the constructor of the object type.
145     template<typename T, typename... Args>
146     gc_ptr<T> make_gc_ptr(Args&&... args) {
147         return gc_ptr<T>(new T(std::forward<Args>(args)...));
148     }
149
150 } }
151
152 #endif
```

Listing C.31: trokis/support/gc_reclaimer.hpp

```
1  //
2  //  gc_reclaimer.hpp
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 15/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8
9  #ifndef trokis_gc_reclaimer_hpp
10 #define trokis_gc_reclaimer_hpp
11
12 namespace trokis { namespace gc {
13
14     // Garbage collection reclamation object. Allows gc_ptrs to
           enqueue their objects and
15     // frees these after a set timeout(default 10 seconds) has
           passed. A future addition would
16     // be implementing a proxy collection scheme, allowing garbage
           to be held only until it is
17     // no longer needed, rather than for a given timeout. This
           would result in memory being
18     // reclaimed more quickly, but would also add more overhead to
           the code(as some sort of
```

```
19      // reference count would have to be maintained). The timeout
            based solution avoids this
20      // overhead at the expense of reclamation delay, and also
            ensures that memory can never be
21      // accidentally held forever.
22      class gc_reclaimer
23      {
24      private:
25          using reclaim_clock = std::chrono::steady_clock;
26
27          struct reclaimer_t {
28              reclaimer_t() : ptr(nullptr), inserted_at(reclaim_clock
                    ::now()), next(nullptr) { }
29              ~reclaimer_t() {
30                  if(ptr) {
31                      deleter(ptr); // Performs actual garbage
                            reclamation.
32                  }
33              }
34
35              void* ptr;
36              std::function<void(void*)> deleter;
37              reclaim_clock::time_point inserted_at;
38              reclaimer_t* next;
39          };
40
41          void gc_worker() {
42              while(!terminating_.load()) {
43                  auto current_item = reclaim_queue_.pop();
44                  if(!current_item) {
45                      // Sleep for one second
46                      std::this_thread::sleep_for(std::chrono::
                            seconds(1));
47                      continue;
48                  }
49                  auto threshold = reclaim_clock::now() - timeout_;
50                  while(current_item->inserted_at > threshold && !
                        terminating_.load()) {
51                      std::this_thread::sleep_for(std::chrono::
                            seconds(1));
52                      threshold = reclaim_clock::now() - timeout_;
53                  }
54                  // RAII will delete for us, we simply pull out
                        items and wait until they can be deleted
55              }
56          }
57
58      public:
59          template<typename TimeUnit = std::chrono::seconds>
60          gc_reclaimer(TimeUnit object_timeout = std::chrono::seconds
                (10)) : timeout_(object_timeout) {
61              gc_thread_ = std::thread(&gc_reclaimer::gc_worker, this
                    );
62          }
63
```

```
64          ~gc_reclaimer() {
65              if(gc_thread_.joinable()) {
66                  terminating_ = true;
67                  gc_thread_.join();
68              }
69          }
70
71          template<typename T, typename Deleter>
72          void reclaim(T* ptr, Deleter deleter) {
73              auto reclaimer = std::make_unique<reclaimer_t>();
74              reclaimer->ptr = ptr;
75              reclaimer->deleter = std::forward<Deleter>(deleter);
76              reclaim_queue_.push(std::move(reclaimer));
77          }
78
79      private:
80          std::atomic<bool> terminating_;
81          std::thread gc_thread_;
82          containers::queue_mpsc<reclaimer_t> reclaim_queue_;
83          std::chrono::milliseconds timeout_;
84      };
85      static gc_reclaimer ___global_reclaimer;
86
87      template<typename T>
88      struct static_default_delete
89      {
90          static void call(void* arg) {
91              T* ptr = reinterpret_cast<T*>(arg);
92              std::default_delete<T>()(ptr);
93          }
94      };
95
96  } }
97
98  #endif
```

Listing C.32: trokis/support/neighbor_table.hpp

```
1   //
2   //  neighbor_table.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 13/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_neighbor_table_hpp
10  #define trokis_neighbor_table_hpp
11
12  #include "trokis/types.hpp"
13  #include "trokis/support/gc.hpp"
14
15  #include <map>
16  #include <cassert>
17  #include <memory>
```

```
18 #include <deque>
19 #include <chrono>
20 #include <mutex>
21
22 #include "trokis/protocols/ipv4/types.hpp"
23 #include "trokis/protocols/ieee802/types.hpp"
24
25 namespace trokis { namespace neighbor {
26
27     // Generic neighbor state machine. Encapsulates the state
          machine used for each neighbor
28     // in a neighbor table, and calls out to the given provider
          every time a packet(or solicitation)
29     // needs to be sent out.
30     template<typename Provider>
31     class generic_neighbor {
32     public:
33         using upper_address_type = typename Provider::
              upper_address_type;
34         using lower_address_type = typename Provider::
              lower_address_type;
35
36         static constexpr std::size_t QUEUE_LIMIT = 1000; // Maximum
               1000 packets queued for transmission.
37
38         generic_neighbor(Provider& provider, device_id devid, const
               upper_address_type& upper_addr)
39         : provider_(provider)
40         , devid_(devid)
41         , upper_address_(upper_addr)
42         , lower_valid_(false)
43         , sent_solicit_(false)
44         { }
45
46         // Tries to deliver a packet to the neighbor. If the lower
              address of the neighbor is
47         // not yet known, we instead enqueue the packet and ask the
               provider to solicit a request.
48         packet_verdict deliver(packet_ptr&& pkt) {
49             auto lower_addr = get_lower();
50             if(std::get<0>(lower_addr)) {
51                 // Output the packet
52                 return provider_.output(std::move(pkt), std::get
                       <2>(lower_addr), std::get<1>(lower_addr));
53             } else {
54                 // Queue the packet and ask for a solicit.
55                 query();
56                 enqueue_packet(std::move(pkt));
57                 return packet_verdict::success;
58             }
59         }
60
61         // Set the lower level addreses of the neighbor. The source
               address must be set too,
```

```
62          // but lower layers(or the provider itself) may of course
                ignore it if they choose.
63          void set_lower_address(const lower_address_type& addr,
                const lower_address_type& src) {
64              std::unique_lock<decltype(lower_addr_lock_)> guard(
                    lower_addr_lock_);
65              lower_valid_ = true;
66              lower_address_ = addr;
67              lower_address_src_ = src;
68              guard.unlock();
69
70              // Empty the queue of packets after dropping the lock
71              empty_queue(addr, src);
72          }
73
74      private:
75          using solicit_clock = std::chrono::steady_clock;
76
77          // Ask the Provider to solicit a request for our upper
                layer address. If a solicit was
78          // sent less than a second ago, we don't send one yet.
79          void query() {
80              using namespace std::chrono;
81              std::unique_lock<decltype(solicit_lock_)> guard(
                    solicit_lock_);
82              auto curtime = solicit_clock::now();
83              if(sent_solicit_) {
84                  if(duration_cast<milliseconds>(curtime -
                        last_solicit_time_).count() < 1000) {
85                      return;
86                  }
87              }
88              sent_solicit_ = true;
89              last_solicit_time_ = curtime;
90              // Drop the solicit lock here so other threads can
                    progress earlier
91              guard.unlock();
92              provider_.solicit(devid_, upper_address_);
93          }
94
95          // Enqueues a packet to be transmitted once the ARP solicit
                 is answered
96          void enqueue_packet(packet_ptr&& pkt) {
97              std::lock_guard<decltype(queue_lock_)> guard(
                    queue_lock_);
98              queue_.emplace_back(solicit_clock::now(), std::move(pkt
                    ));
99              while(queue_.size() > QUEUE_LIMIT) { // Limit the
                    maximum size of the queue
100                 queue_.pop_front();
101             }
102         }
103
104         // Empties the packet queue for this neighbor entry
```

```
105         void empty_queue(const lower_address_type& addr, const
                lower_address_type& src) {
106             using namespace std::chrono;
107             std::lock_guard<decltype(queue_lock_)> guard(
                    queue_lock_);
108             auto curtime = solicit_clock::now();
109             while(queue_.size() > 0) {
110                 // Send out the packets in the queue. If they were
                        enqueued more than
111                 // 500 ms ago, drop them.
112                 auto& front = queue_.front();
113                 if(duration_cast<milliseconds>(curtime - std::get
                        <0>(front)).count() < 500) {
114                     provider_.output(std::move(std::get<1>(front)),
                            src, addr);
115                 }
116                 queue_.pop_front();
117             }
118         }
119
120         // Retrieves the lower level address along with its
                validity field.
121         // OPTIMIZATION: Make this use CAS so the lower address can
                be retrieved without a lock,
122         // making the fast-path lock-less. For now, since the lock
                is extremely short, we will
123         // use this simpler solution.
124         std::tuple<bool,lower_address_type,lower_address_type>
                get_lower() {
125             std::lock_guard<decltype(lower_addr_lock_)> guard(
                    lower_addr_lock_);
126             return std::make_tuple(lower_valid_, lower_address_,
                    lower_address_src_);
127         }
128
129         Provider& provider_;
130         device_id devid_;
131         const upper_address_type upper_address_;
132
133         std::mutex lower_addr_lock_;
134         lower_address_type lower_address_;
135         lower_address_type lower_address_src_;
136         bool lower_valid_;
137
138         std::mutex queue_lock_;
139         // OPTIMIZATION: Change this to a ring buffer rather than a
                deque
140         std::deque<std::pair<solicit_clock::time_point, packet_ptr
                >> queue_;
141
142         std::mutex solicit_lock_;
143         solicit_clock::time_point last_solicit_time_;
144         bool sent_solicit_;
145     };
146
```

```
147     // Provides a neighborhood table backed by the given Provider.
148     //
149     // The provider specifies the upper and lower level addresses
            that it maps
150     // between, and performs this mapping for any device that is
            used with it.
151     template<typename Provider>
152     class neighbor_table {
153     public:
154         using provider_type = Provider;
155         using neighbor_type = generic_neighbor<Provider>;
156         using upper_address_type = typename provider_type::
                upper_address_type;
157         using lower_address_type = typename provider_type::
                lower_address_type;
158 
159         template<typename... Args>
160         neighbor_table(Args&&... args) : provider_(std::forward<
                Args>(args)...)
161         { }
162 
163         neighbor_type* get_or_create(const device_id devid, const
                upper_address_type& addr) {
164             std::lock_guard<decltype(neighbors_lock_)> guard(
                    neighbors_lock_);
165             auto key = std::make_tuple(devid, addr);
166             auto& item = neighbors_[key];
167             if(item.acquire() == nullptr) { // We are under lock,
                    don't need to do CAS
168                 item.assign(std::make_unique<neighbor_type>(
                        provider_, devid, addr).release());
169             }
170             return item.acquire();
171         }
172 
173         neighbor_type* get(const device_id devid, const
                upper_address_type& addr) {
174             std::lock_guard<decltype(neighbors_lock_)> guard(
                    neighbors_lock_);
175             auto key = std::make_tuple(devid, addr);
176             auto retval = neighbors_.find(key);
177             if(retval != neighbors_.end()) {
178                 return retval->second.acquire();
179             } else {
180                 return nullptr;
181             }
182         }
183 
184         void flush() {
185             std::lock_guard<decltype(neighbors_lock_)> guard(
                    neighbors_lock_);
186             neighbors_.clear();
187         }
188 
```

```
189         packet_verdict transmit(packet_ptr&& pkt, const
                upper_address_type& addr) {
190             auto neigh = get_or_create(pkt->dev, addr);
191             if(neigh == nullptr) {
192                 return packet_verdict::dropped;
193             }
194             return neigh->deliver(std::move(pkt));
195         }
196 
197         Provider& get_provider() {
198             return provider_;
199         }
200 
201     private:
202         Provider provider_;
203         std::map<std::tuple<device_id, upper_address_type>, gc::
                gc_ptr<neighbor_type>> neighbors_;
204         std::mutex neighbors_lock_;
205     };
206 
207 } }
208 
209 #endif
```

Listing C.33: trokis/support/pcap_output.hpp

```
1  //
2  //  pcap_dumper.h
3  //  trokis
4  //
5  //  Created by Lasse Bang Dalegaard on 14/07/13.
6  //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7  //
8 
9  #ifndef trokis_pcap_dumper_h
10 #define trokis_pcap_dumper_h
11 
12 #include <fstream>
13 #include <algorithm>
14 #include <array>
15 #include <thread>
16 #include <iterator>
17 #include <mutex>
18 
19 #include "trokis/packet.hpp"
20 #include "trokis/support/byte_tools.hpp"
21 
22 namespace trokis { namespace tools {
23 
24     enum pcap_encap {
25         ethernet = 1,
26         ipv4     = 228,
27         ipv6     = 229
28     };
29 
```

```
30      namespace detail {
31
32          struct pcap_output_wrapper {
33              pcap_output_wrapper(std::string filename, pcap_encap
                    encap_type)
34              : os(filename, std::ios::trunc | std::ios::binary | std
                    ::ios::out), osi(os), encap_(encap_type)
35              {
36                  using namespace trokis::byte_tools;
37                  write_bytes<endianness_host>((uint32_t)0xa1b2c3d4,
                        osi);
38                  write_bytes<endianness_host>((uint16_t)2, osi);
39                  write_bytes<endianness_host>((uint16_t)4, osi);
40                  write_bytes<endianness_host>((int32_t)0, osi);
41                  write_bytes<endianness_host>((uint32_t)0, osi);
42                  write_bytes<endianness_host>((uint32_t)~0, osi);
43                  write_bytes<endianness_host>((uint32_t)encap_, osi)
                        ;
44              }
45
46              template<typename Iterator>
47              void output(const Iterator& begin, const Iterator& end)
                     {
48                  using namespace trokis::byte_tools;
49
50                  std::lock_guard<std::mutex> lock(glob_lock);
51                  auto timestamp = std::chrono::microseconds(std::
                        chrono::seconds(std::time(NULL))).count();
52                  uint32_t length = uint32_t(std::distance(begin, end
                        ));
53
54                  write_bytes<endianness_host>((uint32_t)(timestamp /
                         1000000), osi);
55                  write_bytes<endianness_host>((uint32_t)(timestamp %
                         1000000), osi);
56                  write_bytes<endianness_host>(length, osi);
57                  write_bytes<endianness_host>(length, osi);
58
59                  std::copy(begin, end, osi);
60              }
61
62          private:
63
64              std::ofstream os;
65              std::ostream_iterator<uint8_t> osi;
66              std::mutex glob_lock;
67              pcap_encap encap_;
68          };
69
70      }
71
72      // Helper class for outputting packets to a .pcap file.
73      class pcap_output
74      {
75      public:
```

```
76          pcap_output(std::string filename, pcap_encap encap_type =
                pcap_encap::ethernet)
77          : wrapper_(std::make_shared<detail::pcap_output_wrapper>(
                filename, encap_type)) { }
78
79          pcap_output(const pcap_output& other) : wrapper_(other.
                wrapper_) { }
80
81          pcap_output(pcap_output&& other) : wrapper_(std::move(other
                .wrapper_)) { }
82
83          template<typename... Args>
84          packet_verdict operator()(packet_ptr&& pkt, Args&&... args)
                {
85              if(!pkt) {
86                  return packet_verdict::dropped;
87              }
88              auto& pdata = pkt->data();
89              wrapper_->output(pdata.begin(), pdata.end());
90              return packet_verdict::success;
91          }
92
93      private:
94          std::shared_ptr<detail::pcap_output_wrapper> wrapper_;
95      };
96
97  } }
98
99  #endif
```

Listing C.34: trokis/support/routing_table.hpp

```
1   //
2   //  routing_table.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 12/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_routing_table_hpp
10  #define trokis_routing_table_hpp
11
12  #include <algorithm>
13  #include <vector>
14
15  namespace trokis { namespace routing {
16
17      // Applies the given mask to the address. The address must be
            of a byte array-compatible type.
18      template<typename AddressType>
19      AddressType masked_address(AddressType addr, uint8_t netmask) {
20          uint8_t remaining = netmask;
21          auto cur = std::begin(addr);
22          constexpr auto part_len = sizeof(*cur);
```

```
23          for(auto& cur : addr) {
24              if(remaining == 0) {
25                  cur = 0;
26              } else if(remaining >= 8 * part_len) {
27                  remaining -= 8 * part_len;
28              } else {
29                  // Blank the last `remaining` bits of this section
30                  cur &= ~((1 << (8 * part_len - remaining)) - 1);
31                  remaining = 0;
32              }
33          }
34          return addr;
35      }
36
37      // Performs a longet-prefix-match between a route and and
            address.
38      template<typename AddressType>
39      bool route_match_lpm(AddressType addr, AddressType route,
            uint8_t netmask) {
40          return masked_address(addr, netmask) == route;
41      }
42
43      // Encapsulates a route.
44      template<typename AddressType>
45      struct route {
46          AddressType addr;
47          uint8_t mask;
48          bool direct;
49          AddressType nexthop_addr;
50          device_id dev;
51      };
52
53      // Simple routing table for trokis. Currently simply uses a
            list of routes and sequentially
54      // matches these. This will of course not scale to a large
            amount of routes, but is very simple
55      // and allows great performance for a low amount of routes. The
             routing table itself implements
56      // no locking, meaning that it is up to the client to ensure
            that access to the routing table is
57      // concurrency safe. The recommended approach(and the one used
            in the trokis clients, IPv4 and IPv6)
58      // is to make a complete copy of the routing table, update the
            copy and finally atomically commit the
59      // table using a compare-and-swap. This allows wait-free
            waiting for readers and lock-free access for
60      // writers.
61      template<typename AddressType>
62      class routing_table {
63      public:
64          using route_type = route<AddressType>;
65
66          void add_route(const route_type& route) {
67              routes_.push_back(route);
68              auto& back = routes_.back();
```

```
69              back.addr = masked_address(back.addr, back.mask);
70          }
71
72          void del_route(const route_type& route) {
73              std::remove(std::begin(routes_), std::end(routes_),
                    route);
74          }
75
76          const route_type* find_route_lpm(const AddressType addr)
                const {
77              const route_type* curbest = nullptr;
78              for(auto& route : routes_) {
79                  if(route_match_lpm(addr, route.addr, route.mask)) {
80                      if(curbest != nullptr && curbest->mask > route.
                            mask) {
81                          continue;
82                      }
83                      curbest = &route;
84                  }
85              }
86              return curbest;
87          }
88
89      private:
90          std::vector<route_type> routes_;
91      };
92
93  } }
94
95  #endif
```

Listing C.35: trokis/types.hpp

```
1   //
2   //  types.hpp
3   //  trokis
4   //
5   //  Created by Lasse Bang Dalegaard on 09/07/13.
6   //  Copyright (c) 2013 Lasse Bang Dalegaard. All rights reserved.
7   //
8
9   #ifndef trokis_types_hpp
10  #define trokis_types_hpp
11
12  #include "trokis/detail/compat.h"
13
14  #include <memory>
15
16  namespace trokis {
17
18      using byte_t = uint8_t;
19
20      // Forward declarations
21      class packet;
22      class netdevice;
```

```
23      class worker;
24
25      // Generic packet verdicts
26      enum class packet_verdict {
27          success,        // Packet was received or buffered
28          dropped         // Packet was(should be) dropped
29      };
30
31      // Generic address type information
32      enum class address_type {
33          unicast,
34          multicast,
35          broadcast
36      };
37
38      // Pointer to a packet
39      using packet_ptr = std::unique_ptr<packet>;
40
41      // ID types
42      using worker_id = int;
43      using device_id = int;
44  }
45
46  #include "trokis/support/gc.hpp"
47
48  #endif
```

Bibliography

[1] Dave Evans, Cisco Systems. *The Internet of Things: How the Next Evolution of the Internet is Changing Everything*. White paper. Apr. 2011. URL: http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[2] Eddie Kohler. "The Click Modular Router". PhD thesis. Massachusetts Institute of Technology, 2001. URL: http://pdos.csail.mit.edu/papers/click:kohler-phd/thesis.pdf.

[3] Daniel Borkmann. "Lightweight Autonomic Network Architecture". MA thesis. ETH Zurich, 2012. URL: http://www.epics-project.eu/publications/2012_borkmann_mastersthesis.pdf.

[4] Tufts University. *Perseus - Greek Word Study Tool*. July 2013. URL: http://www.perseus.tufts.edu/hopper/morph?l=tro/xis&la=greek&can=tro/xis0&prior=suna/ggelos.

[5] International Telecommunication Union. *ICT Facts and Figures 2013*. Feb. 2013. URL: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013.pdf.

[6] R. Hinden and S. Deering. *IP Version 6 Addressing Architecture*. RFC 4291 (Draft Standard). Updated by RFCs 5952, 6052. Internet Engineering Task Force, Feb. 2006. URL: http://www.ietf.org/rfc/rfc4291.txt.

[7] Ascom Holding AG. *Protocol Overhead in GPRS*. Sept. 2002. URL: http://www.ascom.com/cn/protocol-overhead.pdf.

[8] European Telecommunications Standards Institute. *Universal Mobile Telecommunications System (UMTS); Packet Data Convergence Protocol (PDCP) specification*. 2012. URL: http://www.etsi.org/deliver/etsi_ts/125300_125399/125323/05.02.00_60/ts_125323v050200p.pdf.

[9] Deepankar Medhi and Karthikeyan Ramasamy. *Network routing - algorithms, protocols, and architectures*. Morgan Kaufmann, 2007. ISBN: 978-0-12-088588-6.

[10] O. Spatscheck. "Layers of Success". In: *IEEE Internet Computing* 17.1 (2013), pp. 3–6. URL: http://www.computer.org/csdl/mags/ic/2013/01/mic2013010003.pdf.

[11] R. Bush and D. Meyer. *Some Internet Architectural Guidelines and Philosophy*. RFC 3439 (Informational). Internet Engineering Task Force, Dec. 2002. URL: http://www.ietf.org/rfc/rfc3439.txt.

[12] S. Frankel and S. Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071 (Informational). Internet Engineering Task Force, Feb. 2011. URL: http://www.ietf.org/rfc/rfc6071.txt.

[13] ISO, Geneva, Switzerland. *ISO/IEC 7498-1, 2nd edition: Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. Nov. 1994. URL: http://www.ecma-international.org/activities/Communications/TG11/s020269e.pdf.

[14] R. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (INTERNET STANDARD). Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. Internet Engineering Task Force, Oct. 1989. URL: http://www.ietf.org/rfc/rfc1122.txt.

[15] B. Carpenter. *Architectural Principles of the Internet*. RFC 1958 (Informational). Updated by RFC 3439. Internet Engineering Task Force, June 1996. URL: http://www.ietf.org/rfc/rfc1958.txt.

[16] F. Baker. *Requirements for IP Version 4 Routers*. RFC 1812 (Proposed Standard). Updated by RFCs 2644, 6633. Internet Engineering Task Force, June 1995. URL: http://www.ietf.org/rfc/rfc1812.txt.

[17] C. Hornig. *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. RFC 894 (INTERNET STANDARD). Internet Engineering Task Force, Apr. 1984. URL: http://www.ietf.org/rfc/rfc894.txt.

[18] J. Postel and J.K. Reynolds. *Standard for the transmission of IP datagrams over IEEE 802 networks*. RFC 1042 (INTERNET STANDARD). Internet Engineering Task Force, Feb. 1988. URL: http://www.ietf.org/rfc/rfc1042.txt.

[19] IEEE. *IEEE Std 802-2001: IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture*. Feb. 2002. URL: http://standards.ieee.org/getieee802/download/802-2001.pdf.

[20] IEEE. *IEEE Std 802.3™-2012*. Dec. 2012. URL: http://standards.ieee.org/getieee802/download/802.3-2012_section1.pdf.

[21] Internet Assigned Numbers Authority (IANA). *IEEE 802 Numbers*. Mar. 2012. URL: http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[22] IEEE. *IEEE Std 802.2, 1998 Edition(R2003)*. May 1998. URL: http://standards.ieee.org/getieee802/download/802.2-1998.pdf.

[23] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-end arguments in system design". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. URL: http://doi.acm.org/10.1145/357401.357402.

[24] FreeBSD kernel developers. *FreeBSD 9.1 kernel source code*. Dec. 2012. URL: http://svnweb.freebsd.org/base/stable/9/sys/.

[25] Linux kernel developers. *Linux 3.10.2 source code*. July 2013. URL: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/log/?id=refs/tags/v3.10.2.

[26] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946. Internet Engineering Task Force, Dec. 1998. URL: http://www.ietf.org/rfc/rfc2460.txt.

[27] J. McCann, S. Deering, and J. Mogul. *Path MTU Discovery for IP version 6*. RFC 1981 (Draft Standard). Internet Engineering Task Force, Aug. 1996. URL: http://www.ietf.org/rfc/rfc1981.txt.

[28] Intel Corporation, Networking Division. *Intel® 82599 10 GbE Controller Datasheet, rev 2.8*. June 2013. URL: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[29] Hans Boehm, Alan Demers, and Mark Weiser. *Boehm garbage collector*. May 2012. URL: http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[30] Thomas E. Hart et al. "Performance of memory reclamation for lockless synchronization". In: *Journal of Parallel and Distributed Computing* 67.12 (2007), pp. 1270–1285. URL: http://csng.cs.toronto.edu/publication_files/159/jpdc07.pdf.

[31] Kathleen Nichols and Van Jacobson. *Controlled Delay Active Queue Management Controlled Delay Active Queue Management*. Jan. 2013. URL: http://tools.ietf.org/html/draft-nichols-tsvwg-codel.

[32] Luigi Rizzo. "Netmap: a novel framework for fast packet I/O". In: *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=2342821.2342830.

[33] ntop. *PF_RING*. July 2013. URL: http://www.ntop.org/products/pf_ring/.

[34] Hideaki Yoshifuji. *iputils 20121221-2*. Dec. 2012. URL: http://www.skbuff.net/iputils/.

[35] Valgrind™ Developers. *Memcheck: a memory error detector*. 2012. URL: http://valgrind.org/docs/manual/mc-manual.html.

[36] Mihai Dobrescu et al. "RouteBricks: exploiting parallelism to scale software routers". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 15–28. ISBN: 978-1-60558-752-3. URL: http://doi.acm.org/10.1145/1629575.1629578.