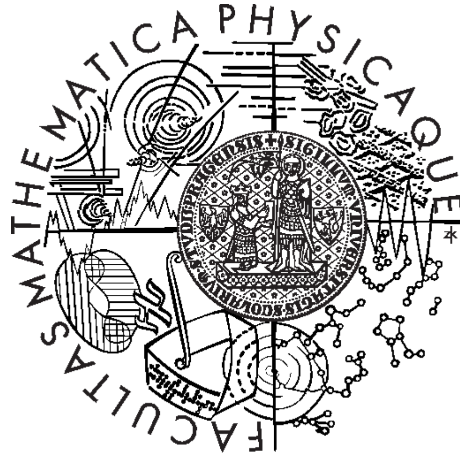
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Mgr. Bc. Antonín Steinhauser

IPv6 for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký
Study programme: Informatics
Specialization: Software Systems

Prague 2013

I am much obliged to my thesis supervisor, Mgr. Martin Děcký, for his advices and hints in this research.

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

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

In Prague date 18. 7. 2013 Antonín Steinhauser

Název práce: IPv6 for HelenOS

Autor: Antonín Steinhauser

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce:

Mgr. Martin Děcký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato práce rozšiřuje operační systém HelenOS o podporu nového IPv6 protokolu. Implementace protokolu IPv6 je na stejné úrovni jako dřívější implementace IPv4 protokolu. Síťový stack HelenOS nyní nabízí tři módy práce se sítí: užívání pouze IPv4 protokolu, užívání pouze IPv6 protokolu a duální mód, který umožňuje používat oba protokoly najednou.

Práce popisuje předchozí stav síťového stacku HelenOS, analyzuje rozdíly mezi IPv4 protokolem a IPv6 protokolem a zdůvodňuje jednotlivá strategická rozhodnutí. Posléze popisuje použité implementační a ladící metody, shrnuje výsledky práce a srovnává HelenOS s jinými mikrojadernými operačními systémy co do podpory IPv6.

Klíčová slova: HelenOS, IPv6, síťový

Title: IPv6 for HelenOS

Author: Antonín Steinhauser

Department: Department of Distributed and Dependable Systems

Supervisor:

Mgr. Martin Děcký, Department of Distributed and Dependable Systems

Abstract: This thesis extends HelenOS operating system in order to be IPv6 capable. New IPv6 protocol implementation is on the same level as previous IPv4 protocol implementation. HelenOS networking stack now offers three modes of networking: IPv4-only, IPv6-only and dual stack mode. Dual stack mode enables usage of both protocols at once.

The thesis describes previous state of HelenOS networking stack, analyzes differences between IPv4 and IPv6 protocols and gives reasons for single strategic decisions. In fine, it describes used implementation and debugging techniques, concludes results and compares HelenOS with other microkernel operating system from the IPv6-capability perspective.

Keywords: HelenOS, IPv6, networking

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Goals . . . 3
1.3 Plan . . . 3
1.4 Contents . . . 3

2 Context **5**
2.1 Introduction to TCP/IP . . . 5
2.1.1 Link layer . . . 5
2.1.2 Network layer . . . 6
2.1.3 Transport layer . . . 7
2.1.4 BSD socket interface . . . 8
2.2 Differences between IPv4 and IPv6 . . . 8
2.2.1 Differences on the link layer . . . 8
2.2.2 Differences on the network layer . . . 9
2.2.3 Differences in network-layer control protocols . . . 12
2.2.4 Differences on the transport layer . . . 13
2.2.5 Differences in BSD socket interface . . . 14
2.3 Introduction to HelenOS . . . 16
2.3.1 HelenOS IPC . . . 16
2.3.2 HelenOS networking stack introduction . . . 17
2.3.3 HelenOS link-layer servers . . . 17
2.3.4 HelenOS network-layer server . . . 19
2.3.5 HelenOS transport-layer servers . . . 23
2.3.6 HelenOS socket library . . . 27

3 Analysis **29**
3.1 Strategic decisions . . . 29
3.1.1 Protocol/task question . . . 29
3.1.2 Data structures common to IPv4 and IPv6 addresses and IPC interface from `inetsrv` "up" . . . 30
3.1.3 NDP position (link-local addresses) and the rest of IPC interface . . . 32
3.2 Interface enhancements . . . 33
3.3 Code upgrades . . . 36
3.3.1 Changes in shared libraries (others than socket library) . . 36
3.3.2 Changes in `ethip` . . . 37
3.3.3 Changes in `loopip` . . . 38
3.3.4 Changes in `inetsrv` . . . 38
3.3.5 Changes in transport-layer servers . . . 41
3.3.6 Changes in socket library . . . 42
3.3.7 Changes in utilities . . . 43

4 Implementation **44**
4.1 Step-by-step development and continuous testing 44
4.2 Debugging methods 44
4.3 Testing set 45
4.3.1 Testing set determination 45
4.3.2 Antecedent bug fixes in previous IPv4 implementation . . 46
4.3.3 Final testing set 47
4.4 Development phases 48
4.4.1 Preliminary phases 49
4.4.2 Porting to IPv6 49

5 Evaluation **54**
5.1 Functional aspects 54
5.2 Performance aspects 55

6 Future & related work **57**
6.1 Future work 57
6.1.1 Scopes support and routing mechanism upgrade 57
6.1.2 Automatic IPv6 address and default router assignment . . 58
6.1.3 Socket library 59
6.1.4 ICMPv6 error messages and MTU discovery 59
6.2 Related work 59
6.2.1 Hurd IPv6 implementation 60
6.2.2 Minix IPv6 implementation 60

7 Conclusion **61**

Bibliography **62**

List of Abbreviations **64**

1. Introduction

1.1 Motivation

Main goal of this thesis is to extend HelenOS networking stack in order to be IPv6 capable. IPv4 addresses are already almost exhausted. IPv6 connectivity gradually starts to be necessary for everyone who wants to use full-value Internet services. That applies also to HelenOS and to its networking capability.

1.2 Goals

Detailed study of current IPv4 implementation is crucial because HelenOS already has a passable IPv4 implementation. IPv4 and IPv6 are just two versions of the same protocol though they are mutually incompatible. Both IP protocols have almost the same purposes. Therefore, if we take current IPv4 implementation and implement all their features in IPv6, we must get a passable IPv6 implementation too. Usually, such features can be covered exactly the same way as in IPv4. Sometimes it is necessary to use analogical technology providing the same feature (e.g. ARP will be replaced with a part of NDP). Features existing in both IPv4 and IPv6 and lacking in current IPv4 implementation are naturally optional in IPv6 implementation too. The only question is what to do with IPv6-exclusive features. Such features include dual-stack sockets, router solicitations, and router advertisements, address scopes, etc. They could not be evidently included in current IPv4 implementation, but they may belong to the reasonable subset of IPv6 protocol. That subset should be covered already by the prototype implementation. There might be also some IPv4-only features omitted in basic IPv6 implementation preserving all IP protocol purposes. Among such features belong for example checksum calculation and verification in IP datagram headers.

1.3 Plan

In the first phase, we will analyze analogies and differences between IPv4 and IPv6 protocols. In the second phase, we will study current IPv4 implementation in HelenOS. In the third phase, we will decide how to implement IPv6 protocol covering all its necessary features. The implementation should cover even maximum of IPv6-exclusive features and should be easily extendable in the future.

1.4 Contents

Chapter *Context* describes basics of TCP/IP architecture and its particular layers. It precisely analyzes differences between IPv4 and IPv6 protocols and explains basics of non-trivial HelenOS multiserver networking stack. Chapter *Analysis* determines necessary features of prototype implementation and gives reasons for single strategic decisions. Chapter *Implementation* introduces implementation strategy and tactics, debugging methodology and bug fixing of previous IPv4 implementation. Chapter *Evaluation* describes what properties the new

implementation has from the functional and efficiency points of view. Chapter *Future and related work* offers future extensions and enhancements of the prototype implementation. It lightly describes also methods how to reach them. Afterwards it compares HelenOS IPv6 implementation with other well-known microkernel operating systems. Finally, chapter *Conclusion* sums this thesis up.

2. Context

2.1 Introduction to TCP/IP

Networking architecture called TCP/IP defines multiple layers of networking. This architecture has had in recent decades no alternatives among computer networking technologies. Each of its layers uses the layer underneath and provides something to the upper one. Data units of the upper one are encapsulated into data unit payloads of the lower one. The layer underneath can send even some other data units without payload from the upper one. Those auxiliary data units are good to assure or enhance its functionality and efficiency (ARP, ICMP, ICMPv6, etc.).

2.1.1 Link layer

The lowest layer is the link layer. Prevalent link-layer protocol is the Ethernet. HelenOS did not support any other link-layer protocols except Ethernet and loopback. Loopback emulates a virtual network interface purely by software. Any frame going to the output returns back unchanged as its input. During implementation of this thesis, even Slip link-layer protocol support was added to HelenOS. Transfer unit of Ethernet is called Ethernet frame. It is composed of source MAC address (6-byte link-layer address assigned to a certain NIC), destination MAC address, type of higher protocol and the payload. Payload is a serialized datagram of the nested protocol. It can be a network-layer protocol such as IP or an auxiliary link-layer protocol such as ARP.

Some MAC addresses have a special semantics. The first special address is broadcast (address with all 48 bits true - `FF:FF:FF:FF:FF:FF`). Datagrams addressed to this special address are received by all NICs connected to Ethernet network. Multicasting is good to afflict only a part of Ethernet nodes, but not only one. Ethernet provides many so-called multicast addresses. Any MAC address with the first byte odd is multicast. The rest of addresses (addresses with the first byte even) are unicast. Unicast addresses can be assigned to one concrete NIC. Only computers connected to the same local network can communicate over Ethernet.

Broadcasts and multicasts are essential for Ethernet functionality. These addresses are used for IP to MAC address translation. The protocol translating IPv4 addresses to MAC addresses is called ARP (Address resolution protocol). Translating node sends a query carrying its IPv4 address from its unicast MAC address to broadcast MAC address. All nodes connected to the same network receive it. The node with asked IPv4 replies from its unicast MAC address to unicast sender MAC address. The later communication between the two nodes is unicast from one to one MAC address. All the other nodes just ignore that communication (usually already the NIC firmware).

IPv6 communication over Ethernet does not use ARP. It was replaced by NDP (Neighbor discovery protocol). NDP is not a part of the link layer but it belongs to the network layer, concretely to ICMPv6 protocol. The principle of NDP translation (IPv6 to MAC) is almost the same as how ARP translates IPv4 to MAC.

2.1.2 Network layer

Network layer is just above the link layer. Network-layer protocol is Internet protocol (IP). This protocol completely constitutes the Internet. It assigns to any computer worldwide ideally unique IP address. IP protocol contains many relatively complicated mechanisms how to route its data units (IP datagrams) through the whole network (routing tables, subnets, two layers of routing, etc.).

To deliver an outgoing or forwarded datagram, OS networking stack checks its destination. If the destination belongs to any of its networks (networks immediately accessible over link-layer protocol), OS sends the datagram directly over link-layer to that destination. Otherwise, it finds the most specific route and sends the datagram over link-layer to that router. The router forwards it successively up to the destination. The least specific route is called default route. Its router accepts all datagrams whose targets are directly unreachable and no better route does exist. This mechanism ensures final delivery of all correctly addressed datagrams. Routing table can be edited manually or automatically. Automatic routing table setting is performed by an address solicitation protocol (DHCP or NDP) or by a specialized routing protocol (OSPF or BGP).

IP protocol is principally unreliable. Any transport protocol over IP claiming reliability (e.g. TCP) must provide additional mechanisms to ensure it (packet numbering, acknowledgements, retransmissions, checksums, etc.). IP protocol provides only best effort communication. Some datagrams may not be delivered at all, some may be delivered damaged, some may be delivered in incorrect order (later sent packet is delivered earlier) and some may be delivered even multiple times.

This Internet protocol has nowadays two mutually incompatible versions - IPv4 and IPv6. It means there are two de-facto independent internets. Version 4 is used since early eighties. This version is still prevalent but it has many insufficiencies. Firstly, its address space is already practically exhausted and there is no regular possibility how to connect more computers. IPv4 addresses are 32-bit so their theoretically maximal count is 2^{32}. IPv4 addresses were even wasted in the past due to inappropriate network sizes (in particular 8-bits A-classes networks were assigned to organizations unable to utilize at least a chief of them).

IPv4 datagram header has variable length and contains many obsolete items (such as *Type of service*). IPv4 protocol widely enables fragmentation. Datagram can be fragmented not only by the sender, but also by computers routing the datagram. Loss of one of those fragments implies whole datagram loss. It is unable to reassemble original datagram without the missing part. Fragmentation

in general is a negative phenomenon and IPv6 practically eliminated it. The only exception is tunneling IP datagrams in IP datagrams. Because of tunneling overhead, tunnel MTU must be lower than MTU of the encapsulating protocol. That is why tunnel MTU cannot reach those 1500 bytes. Next disadvantage of IPv4 consists in unceasing checksum recalculation. Every IP datagram contains one-byte Time to live (TTL) value (in IPv6 named Hop limit). Every routing node decrements datagram TTL and datagrams with zero TTL are discarded to prevent cyclical routing and congestion of the net. Next item of IPv4 header (in IPv6 missing) is header checksum. Each router must at first decrease the datagram TTL value and then recalculate its checksum. It is naturally performance wasting.

Eventually, there were many attempts how to solve or bypass mentioned IPv4 defects. Among the others NAT and subnetting are worth noting. Those technologies enlarged life cycle of IPv4 protocol. Nevertheless, they are also exhausting their limitations. The only sustainable solution is IPv6 transition.

IP datagram header contains source and destination address (in IPv4 4-byte, in IPv6 16-byte), time-to-live counter, type of higher-layer protocol (in IPv4 called protocol, in IPv6 called next header) and payload length. There are even some other not very important items. Time-to-live counter is decremented on each router. It avoids eternal looping of the datagram in badly routed networks. Datagram header is followed by datagram body containing a transport protocol packet or a control protocol packet.

Control protocol for IPv4 protocol is called ICMP. ICMP is widely used for reachability and unreachability detection in IPv4 networks. IPv6 has a new protocol called ICMPv6 summing up ICMP, ARP, DHCP and IGMP functionality inside IPv6 networks.

2.1.3 Transport layer

Transport layer is just above the network layer. Prevalent protocols on the transport layer are UDP and TCP. Both are supported by HelenOS. There are even some other transport layer protocols but HelenOS does not yet support any of them. Both UDP and TCP protocols provide ports. Ports are 2-byte number identifiers of concrete service on a given computer. UDP ensures non-reliable, datagram-based and stateless communication. TCP ensures reliable, stream-based and stateful communication. Both TCP and UDP packets have a header containing two ports (source and destination), checksum and payload length. TCP packet header contains additionally many other items to ensure its functionality (above all the reliability). Each TCP packet has a sequence number, some flow and congestion control data and flags marking half-open, open, closing or broken connection. TCP and UDP packet bodies contain data from the application layer. Application layer is the highest layer of TCP/IP architecture. TCP protocol offers classical streams such as files or pipes. UDP packets are typical messages.

2.1.4 BSD socket interface

User applications can take those two protocols (TCP and UDP) by widely used BSD socket interface. BSD sockets have actually no alternatives among network communication instruments. BSD socket interface defines socket as a basic communication medium. It represents one side of the communication pipe and the communication partner holds the other side. This conception is partially broken by UDP and datagram-based communication. UDP socket can receive datagrams from various sources and send them to various destinations.

BSD socket interface defines a set of functions with well-defined semantics. Applications can use that interface and be completely unaware of network implementation details. HelenOS provides only a subset of BSD socket interface. Applications communicating over UDP can use *socket*, *bind*, *sendto*, *recvfrom* and *closesocket* functions. Application communicating over TCP can use *socket*, *bind*, *listen*, *accept*, *connect*, *send*, *recv* and *closesocket* functions.

2.2 Differences between IPv4 and IPv6

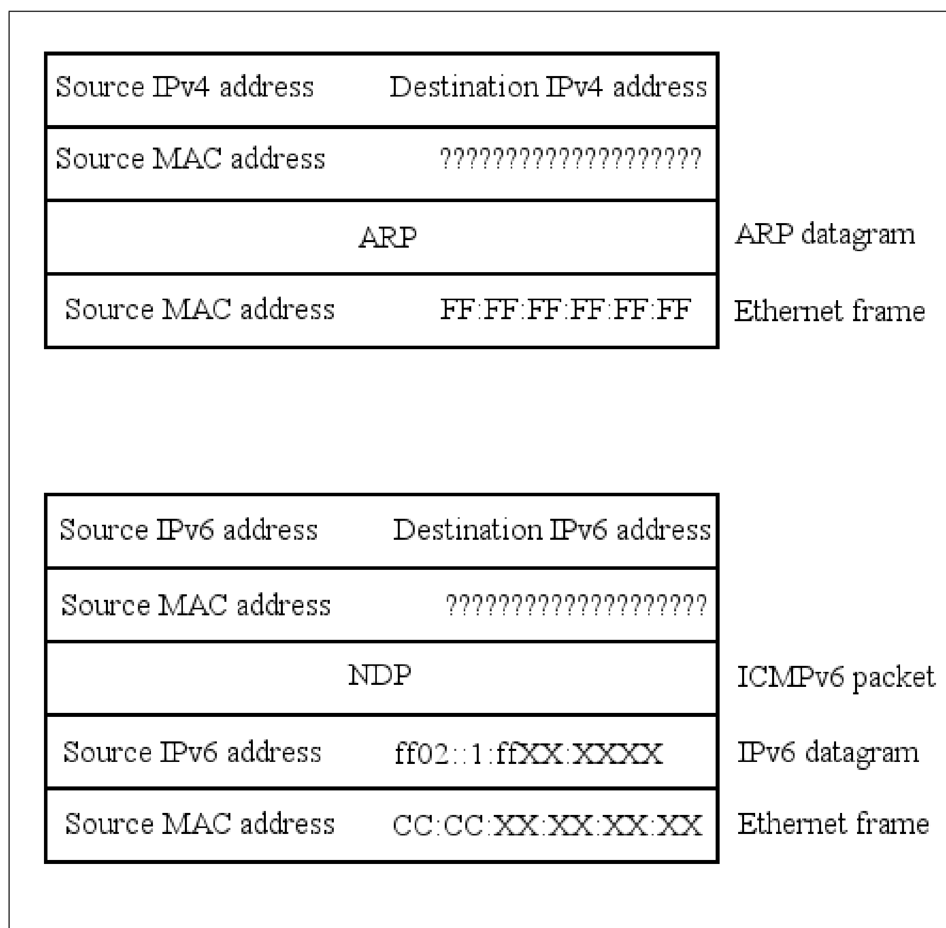
This section describes differences between IPv4 and IPv6 protocols. Although IPv4 and IPv6 are network-layer protocols, they interfere more or less with all TCP/IP layers and protocols.

2.2.1 Differences on the link layer

The mayor difference between sending IPv4 and IPv6 datagrams over Ethernet consists in completely different MAC address resolution. Translation from IPv4 to MAC address is provided by ARP protocol as part of the link layer. Translation from IPv6 to MAC address is provided by NDP protocol. NDP protocol is a subset of ICMPv6 protocol and it belongs to the network layer.

If the OS networking stack does not know MAC address of an IPv4 destination or router, it sends an ARP request. ARP datagram is a special payload of an Ethernet frame not belonging to IPv4 nor to IPv6. It has two possible opcodes - ARP request or ARP reply. ARP request for an unknown MAC address always uses broadcast MAC target.

If the networking stack wants to resolve MAC address of an IPv6 destination or router, it sends a *Neighbor solicitation* message. It is just an ICMPv6 message encapsulated in a common IPv6 datagram. Destination IPv6 address of such datagram is derived from the original target IPv6 address. First 13 bytes are `ff02::1:ff` and the last three bytes are equal to the last three bytes of the original target IPv6 address. Very similar mechanism is used to assign target MAC address for such solicitation. First 2 bytes are constantly `0xCC` and the last 4 bytes are equal to last 4 bytes of the original target IPv6 address. NDP therefore never uses Ethernet broadcasting but only multicasting.



1. **ARP versus NDP**

Additional difference on the link layer is the upper protocol value (ether-type) in Ethernet frame header. Ethernet frames carrying IPv4 datagrams have this value set to `0x0800`, frames carrying ARP datagrams to `0x0806` and frames carrying IPv6 datagrams to `0x86DD`.

2.2.2 Differences on the network layer

Mayor differences between IPv4 and IPv6 protocols occur naturally on the network layer. First difference is the address length. IPv4 addresses have 4 bytes. IPv6 addresses have 16 bytes. There are even many other differences between IPv4 and IPv6 datagram headers.

Version field is set to `4` in IPv4 and to `6` in IPv6 datagrams. *IHL* field from IPv4 is obsolete and absenting in IPv6 datagrams. IPv6 header has fixed length of 40 bytes although additional optional headers can follow. IPv4 header length in 32-bit words is stored in the *IHL* field. Minimal and usual value of *IHL* field is `5`. It means the header has 20 bytes. Optional headers of IPv6 datagrams are stored contrarily as linked list. The *next header* value may not contain directly the code of a transport protocol. It may contain also code of an optional header. Each optional header contains again a *next header* value pointing to another optional

header or to a transport protocol. The last optional header points always to a transport protocol.

Type of service field was also removed. This field is obsolete even in IPv4. IPv4 nodes set it usually to `0` and ignore its value.

Fragmentation parameters *identification*, *flags* and fragment *offset* were moved from the compulsory part to a special optional header called *fragmentation* (code `44`).

Fragmentation is a technique how to pass datagrams longer than MTU of the link layer. IPv6 forbids sending datagrams larger than 1500 bytes unless all nodes on the path explicitly permitted to do so. IPv6 requires MTU at least 1280. Link layer that has MTU less than 1280 is not IPv6 capable. The margin (between 1280 and 1500 bytes) is necessary for tunneling protocols encapsulating network datagrams into network datagrams such as 6to4 or Teredo. When a router forwards an IPv4 datagram over a link with MTU lower than its size, it breaks the datagram into fragments (if the *DF* flag is not set). When it forwards an IPv6 datagram in the same situation, it must discard the datagram and send back an ICMPv6 *Packet too big* message. The sender must react to this message with decreasing the datagram size and sending it again. After the destination receives all necessary fragments, it reassembles the original datagram. IPv6 routers can thus never fragment a datagram. Only the sender has the possibility to do so. The fragmentation in IPv6 should be generally avoided anywhere possible.

Total length field in IPv4 is replaced by *payload length* field in IPv6 with a little different semantics. *Total length* is length of the whole datagram including its header. *Payload length* is its length without those 40 bytes of fix-length header.

Time to live field in IPv4 datagram header was just renamed to *hop limit* in IPv6 datagram header. It is decreased always by one not attending to the time. Nowadays, *time to live* in IPv4 datagrams has in fact the same semantics. One second is a terribly long portion of time for present-day computers and networks.

Protocol field from IPv4 was renamed to *next header* in IPv6. Its semantics was enhanced. Except encapsulated packet type, it may contain an optional header code. Each optional header has therefore again a *next header* field chaining another optional header or finally defining nested packet type as described above.

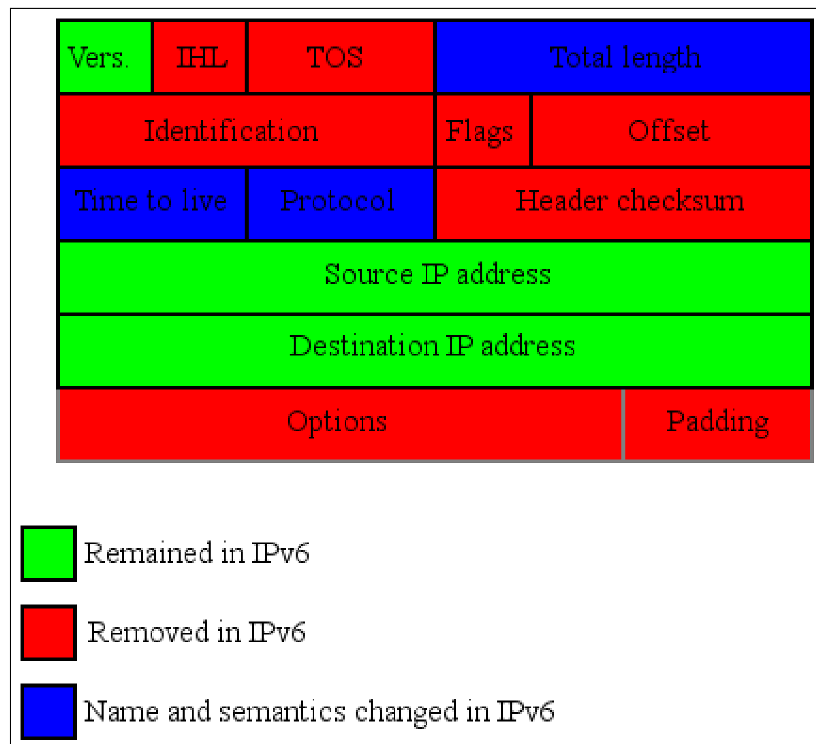
Checksum field used in IPv4 was removed in IPv6. It was not necessary because majority of link layer protocols contain error-detection mechanisms. Similarly, all transport layer protocols (TCP, UDP and ICMPv6) have own checksums in their headers. Those checksums cover also so-called pseudoheaders containing important values copied from IPv6 headers. This additional integrity check on the network layer was redundant and unwanted due to its recalculation on each router as explained above.

Finally, both *source* and *destination addresses* are 4 bytes long in IPv4 datagram headers and 16 bytes long in IPv6 datagram headers.

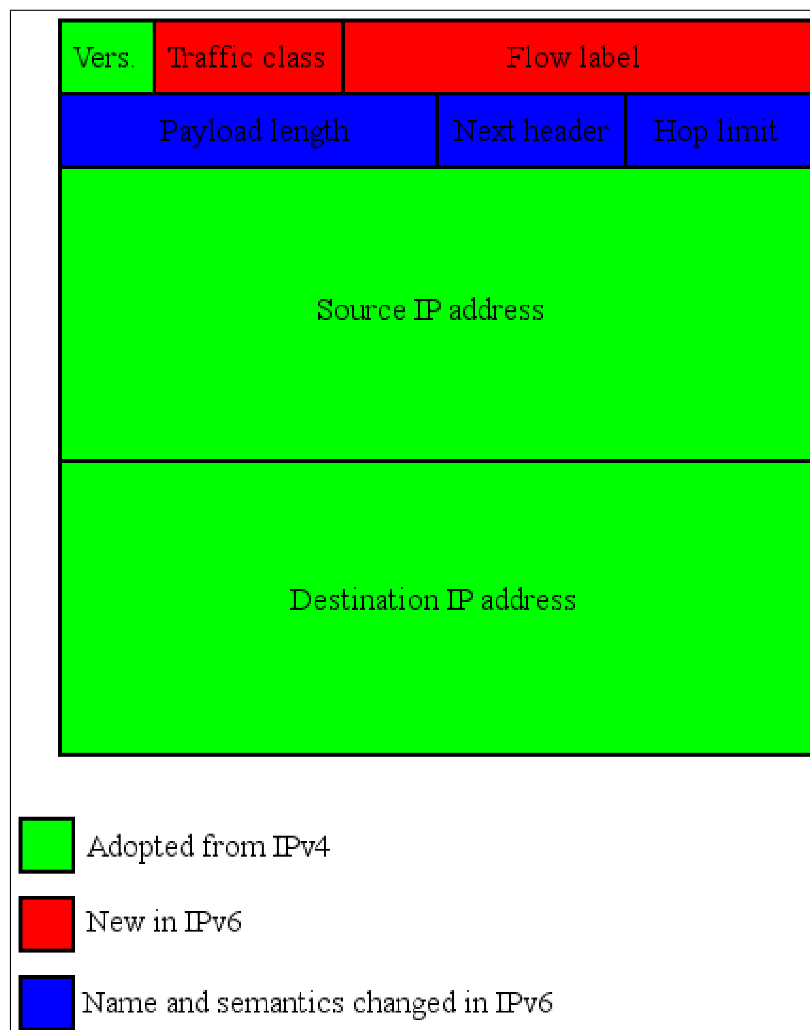
New fields in IPv6 header are *traffic class* and *flow label*. Both are used for QoS and real-time applications. They have no wider usage and majority of their bits are meanwhile reserved.

Optional headers of IPv6 can contain hop-by-hop options, fragmentation, destination options, routing and some other advanced features. Its support is usually optional and they are not necessary for basic IPv6 functionality.

2. IPv4 datagram header



3. IPv6 datagram header



2.2.3 Differences in network-layer control protocols

ICMP protocol, as a part of IPv4 protocol, was upgraded to ICMPv6 simultaneously with IPv4 to IPv6 upgrade. ICMPv6 is not only a new version of ICMP. It is a completely new protocol. ICMP has code `1` and ICMPv6 has code `58`. ICMPv6 is incompatible with IPv4 as well as ICMP is incompatible with IPv6. ICMPv6 header is similar to ICMP header. It contains also one byte of *type*, next byte of *code* and two bytes of *checksum* followed by something what is dependent on the concrete message *type*. The semantics of *checksum* is a little different. ICMP *checksum* covers the ICMP packet only. ICMPv6 *checksum* covers ICMPv6 packet with an artificial pseudoheader containing some items from the IPv6 header. Those items are precisely specified in the next section. The ICMPv6 checksum pseudoheader is equal to TCP/UDP over IPv6 checksum pseudoheader.

Types of ICMPv6 messages are also different to ICMP messages with the same or similar purpose. ICMP *Echo request* message has ICMP type `8` and *Echo*

reply message has ICMP type 0. ICMPv6 *Echo request* has ICMPv6 type 128 and *Echo reply* has ICMPv6 type 129.

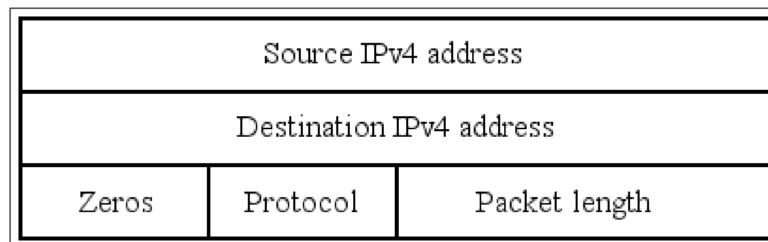
The most significant innovation of ICMPv6 related to HelenOS is the integration of previous ARP functionality - the Neighbor discovery protocol (NDP) as mentioned above. ARP request is replaced with *Neighbor solicitation* message (ICMPv6 type 135). ARP reply is replaced with *Neighbor advertisement* message (ICMPv6 type 136).

ICMPv6 integrates even some other once independent protocol functionality such as IGMP. Worth noting is DHCP replacement with *Router solicitation* (type 133) and *Router advertisement* (type 134) messages. Stateless mechanism of IP address and default router assignment is one of the new IPv6 protocol features.

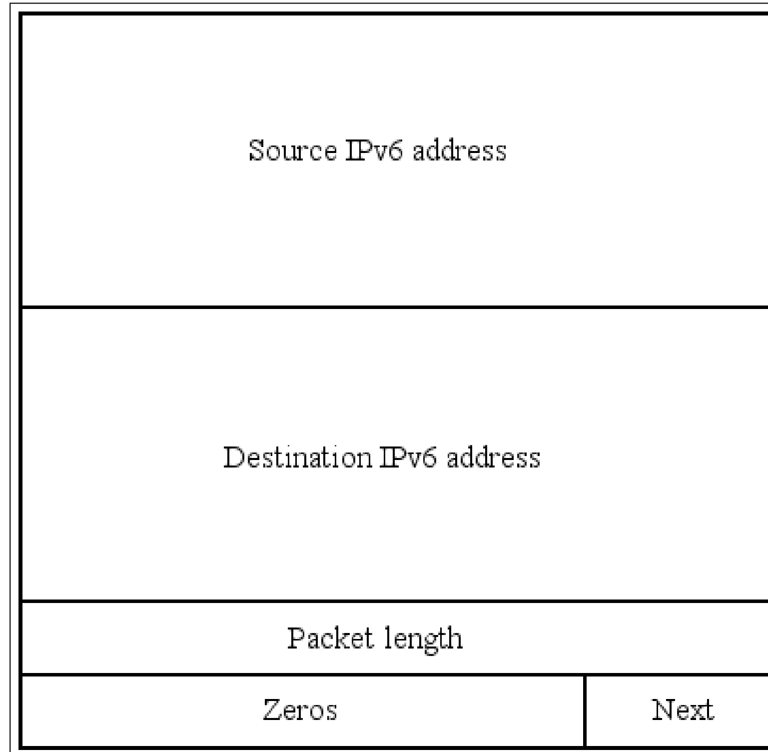
2.2.4 Differences on the transport layer

Transport layer protocols are affected with the IPv6 transition less than network-layer protocols. The only basic change is different TCP and UDP checksum calculation. Both TCP and UDP packet headers contain *checksum* field. In UDP, the *checksum* calculation is optional and its value may be set to zero, but almost every implementation including HelenOS calculates it. The *checksum* covers not only the packet but also a pseudoheader containing some values from the IP datagram header. This pseudoheader is in IPv6 different because *source* and *destination* addresses have different lengths and the packet *length* field is expanded from two to four bytes.

4. TCP or UDP over IPv4 checksum pseudoheader



5. **TCP, UDP or ICMPv6 over IPv6 checksum pseudoheader**



2.2.5 Differences in BSD socket interface

BSD sockets are a very generic instrument. BSD sockets are useful not only for TCP and for UDP communication. They are useful also directly on the network layer (so-called raw sockets) or even on the link layer (for receiving and sending manually constructed Ethernet frames). HelenOS supports by this time only TCP and UDP sockets. That is why only differences between sockets for UDP/TCP over IPv4 and sockets for UDP/TCP over IPv6 are to be explained.

Firstly, there is a new family of sockets. Instead of *AF_INET* for IPv4 communication, it is *AF_INET6* family.

IPv4 and IPv6 protocols are mutually incompatible. However, socket interface ensures backward compatibility of *AF_INET6* sockets with IPv4 protocol. *AF_INET* sockets can never be used for IPv6 communication but *AF_INET6* sockets are useful also for IPv4 communication. It is very important especially for dual stack servers. Dual stack servers are either TCP servers listening on both protocols (e.g. webserver) or UDP servers receiving both IPv4 and IPv6 messages (e.g. DNS server).

This mechanism is called IPv4-mapped IPv6 address. It is an IPv6 address with first 10 bytes (indexes 0 - 9) set to zero, next two bytes (indexes 10 and 11) to `0xFF` and the last four bytes (indexes 12 - 15) are equal to the four bytes of mapped IPv4 address. Such an address is dealt similarly as if it was IPv4 address.

It is returned also from *accept* or *recvfrom* functions when an *AF_INET6* socket accepts an IPv4 connection or receives an IPv4 message.

Socket address structure is a little different. The first item called *sin_family* resp. *sa_family* or *sin6_family* is set to *AF_INET6*. Both *sockaddr_in* and *sockaddr_in6* structures get casted to generic *sockaddr* structure supported by BSD socket interface. The *sa_family* value is the only possibility how to differentiate between socket address families and cast the pointer back to *sockaddr_in* or to *sockaddr_in6*. TCP or UDP port is only renamed (from *sin_port* to *sin6_port*). IP address is larger and it is expressed as a char-array. New item *sin6_flowinfo* helps to specify *flow label* field in IPv6 datagram headers.

6. Socket address for TCP or UDP over IPv4

```
struct sockaddr_in {
    uint16_t sin_family = AF_INET;
    struct in_addr sin_addr;
    uint16_t sin_port;
};

struct in_addr {
    uint32_t s_addr;
};
```

7. Socket address for TCP or UDP over IPv6

```
struct sockaddr_in6 {
    uint16_t sin6_family = AF_INET6;
    struct in6_addr sin6_addr;
    uint16_t sin6_port;
    uint32_t sin6_flowinfo;
};

struct in6_addr {
    uint8_t s6_addr [16];
};
```

2.3 Introduction to HelenOS

HelenOS is a microkernel, multiserver and general-purpose operating system written from scratch[1].

Microkernel operating systems are minority compared to monolithic operating systems such as Linux, Microsoft Windows, majority of BSD systems, etc. Microkernel must be as small as possible. Usually it provides only handlers and calls for CPU privileged instructions and operations (I/O access, physical memory management, interrupt handling, etc.), thread scheduling and inter-process communication (IPC). Everything else should run in userland.

HelenOS processes are called tasks. Each task can have multiple threads. Like any other operating system, HelenOS provides some services for user applications. Those services are called servers. Servers, again tasks, are de-facto another common user applications. That is why HelenOS is called multiserver. Some servers are privileged tasks exercising privileged operations but running in unprivileged CPU mode. One of these servers is `nic` - network interface controller. It interacts directly with the hardware. The other networking stack servers are unprivileged.

Microkernels have many advantages but also disadvantages. One of the major advantages is better stability because any userland application can be killed and eventually restarted without system shutdown and reboot. Fatal problems can thus arise theoretically only from the kernel code. Kernel code should be small enough to be carefully tested and debugged. Among disadvantages belongs higher number of context switches than in monolithic kernels. It causes that process scheduling takes up relatively higher portion of CPU time.

2.3.1 HelenOS IPC

Inter-process communication in microkernel OS is much more important than in monolithic OS. Prevalent service supplying in monolithic operating systems goes through system calls. The communication is hierarchical between an unprivileged application and the privileged kernel. Microkernel operating systems provide majority of their services through IPC. It means two more-less similar applications communicate horizontally. The kernel only provides the connection and mediates their communication. Only if one application unexpectedly terminates, the kernel handles the connection and correctly informs the other.

HelenOS provides standard message passing framework extended with larger data block sending and receiving and with memory sharing possibilities. Sending task passes a message and eventually (or even immediately) waits for its reply. Receiving task picks up this message and processes it to send the reply. Standard messages can contain up to 5 integer values in both directions. Limit for larger data blocks is by then 64 kB. It is large enough to fit all frames, datagrams and packets. Memory sharing is thus not used anywhere in the networking stack.

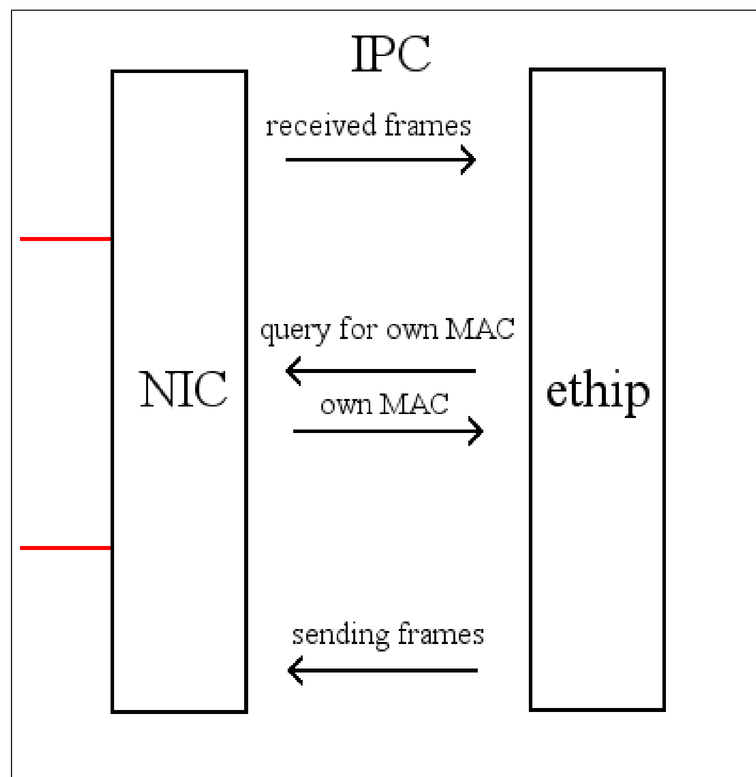
[1]HelenOS, on-line at `http://www.helenos.org/`

Interface of IPC is relatively user-friendly thanks to HelenOS asynchronous library. IPC library creates a fibril for each inter-process connection. Fibrils are user-space threads, smallest chains of code execution. Difference between threads and fibrils is that the kernel schedules threads and is unaware of fibrils. Fibrils enable parallelization of such calls. The state-automata work is encapsulated in the set of IPC functions.

2.3.2 HelenOS networking stack introduction

HelenOS networking stack runs completely in userland. It consists of six servers and one public socket library. The only server interacting directly with the hardware is `nic` - network interface controller. It receives and sends Ethernet frames and can return own MAC address. It is a device driver and is viewed as a black box from the point of view of this thesis.

8. IPC between `nic` and `ethip`



2.3.3 HelenOS link-layer servers

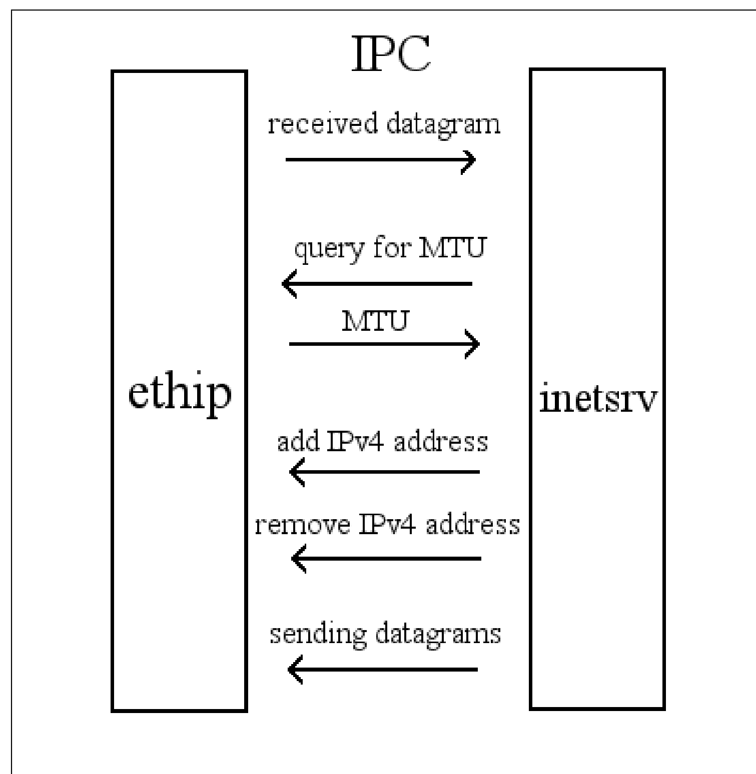
`Ethip` task communicates over IPC with `nic`. It implements Ethernet protocol and stores all Ethernet network interfaces with their IPv4 and MAC addresses. `Ethip` implements also the ARP protocol because it is logically tight up with the Ethernet.

Each received Ethernet frame is differentiated by higher protocol type. If the type is ARP, `ethip` processes it by itself. If it carries an IP datagram, the datagram is sent over IPC to `inetsrv` process.

`Ethip` receives IP datagrams over IPC from `inetsrv`. It must resolve MAC address of the target by ARP protocol. Firstly, it looks into ARP cache. If the MAC address is cached there, it just packs up the datagram into Ethernet frame and sends it. Otherwise, it sends an ARP request for the destination MAC address and the sending fibril waits on a conditional variable. If the ARP reply arrives in one second, ARP processing thread wakes the sending fibril and the datagram is packed and sent. If the ARP reply does not come betimes, IP datagram is discarded.

`Inetsrv` passes to `ethip` every IPv4 address setting when an address is added to or removed from an Ethernet interface. `Ethip` needs such information for correct ARP protocol functionality. `Ethip`, when asked, can also return MTU of Ethernet link. It is now hard-coded to standard minimal value `1500`.

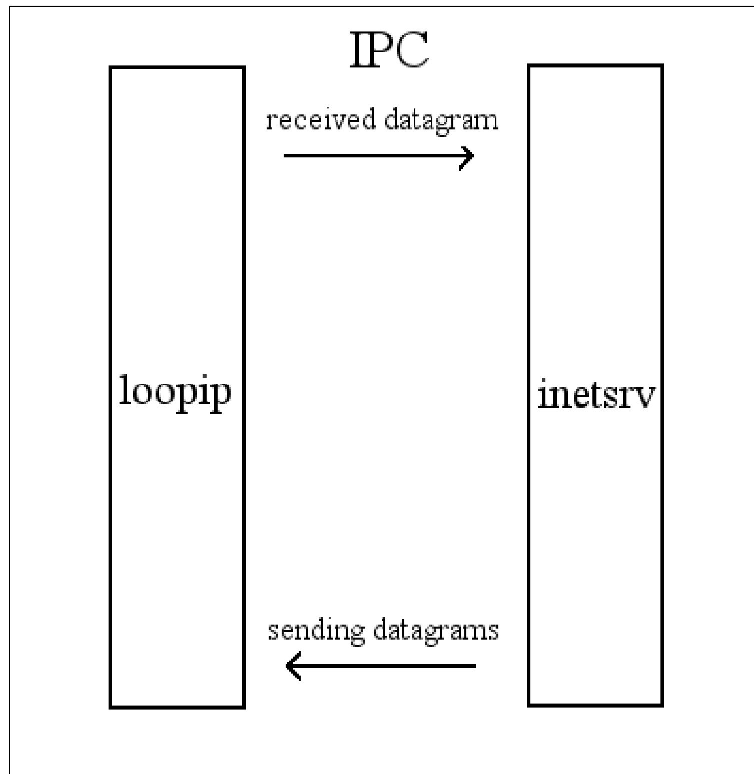
9. IPC between `ethip` and `inetsrv`



An alternative link-layer server to `ethip` is `loopip`. It represents loopback interface returning any outgoing IP datagram back as incoming. Its interface is much easier. There is no need of MTU queries, because its MTU can be arbitrarily increased. Whole interface is emulated purely by software and has nothing to do with any hardware limitations (perhaps except heap memory allocation). `Loopip`

does neither need to know any IP addresses because there are no MAC addresses and no ARP protocol.

10. IPC between `loopip` and `inetsrv`



2.3.4 HelenOS network-layer server

`Inetsrv` server implements the IP protocol. It communicates over IPC, as client, with link-layer servers (`loopip` and `ethip`). As a server, it provides three methods of IPC communication (IPC ports) differentiated by the aim of the second-side application: `inet`, `inetcfg` and `inetping`.

`Inetsrv` holds list of all network interfaces with their link-layer servers - `ethip` or `loopip`. For each network interface, it stores all IP addresses assigned to it together with their network masks. It stores routing table as a list of triplets (network IP address, network mask, router IP address). Each network interface, IP address or routing table entry has its unique name for better manipulation through IPC configuration interface.

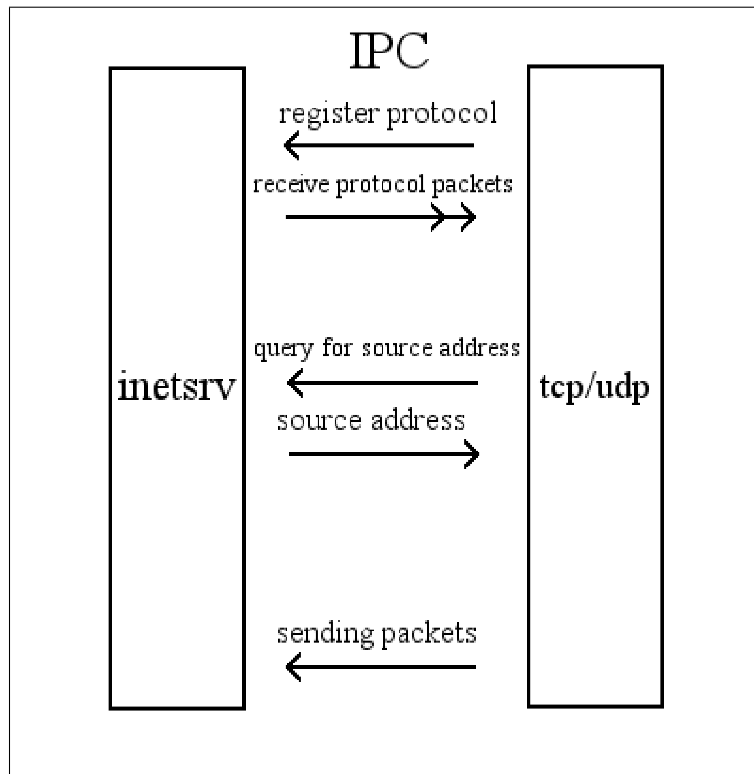
First IPC port (`inet`) is the most important. Transport protocol servers (`tcp` and `udp`) use it for communication with `inetsrv`. Each transport protocol after connecting to `inetsrv` registers its protocol number. It must be exactly the same number which appears in IP headers in *protocol* or *next header* field - `TCP = 6`, `UDP = 17`.

This registration is necessary for `inetsrv` to deliver incoming packets of this protocol correctly. When `inetsrv` receives an IP datagram (by IPC from a link-layer server), it checks the protocol number. If the protocol number is ICMP, `inetsrv` processes it by itself. Otherwise, it looks to the list of registered protocols, extracts the packet from IP datagram and sends it with both source and destination IP addresses over IPC to matching transport protocol server. If `inetsrv` does not find such a protocol, it automatically discards the received datagram.

Transport protocol servers can also ask for source IP address in order to be used for a given destination IP address. `Inetsrv` checks if the destination address is reachable directly from a link-layer interface. If the address is directly reachable, `inetsrv` returns own address in the same network as the destination. Otherwise, it searches the routing table and returns own address in the same network as the best router.

Finally, transport protocols can send outgoing packets. They must specify datagram source and destination IP addresses and naturally the payload. `Inetsrv` encodes IP datagram and looks where to send it. If the destination address belongs to network of an interface, `inetsrv` chooses that interface. Otherwise it looks to the routing table for the most specific router and chooses its interface. In the end, `inetsrv` sends the datagram over IPC to the link-layer server of chosen interface.

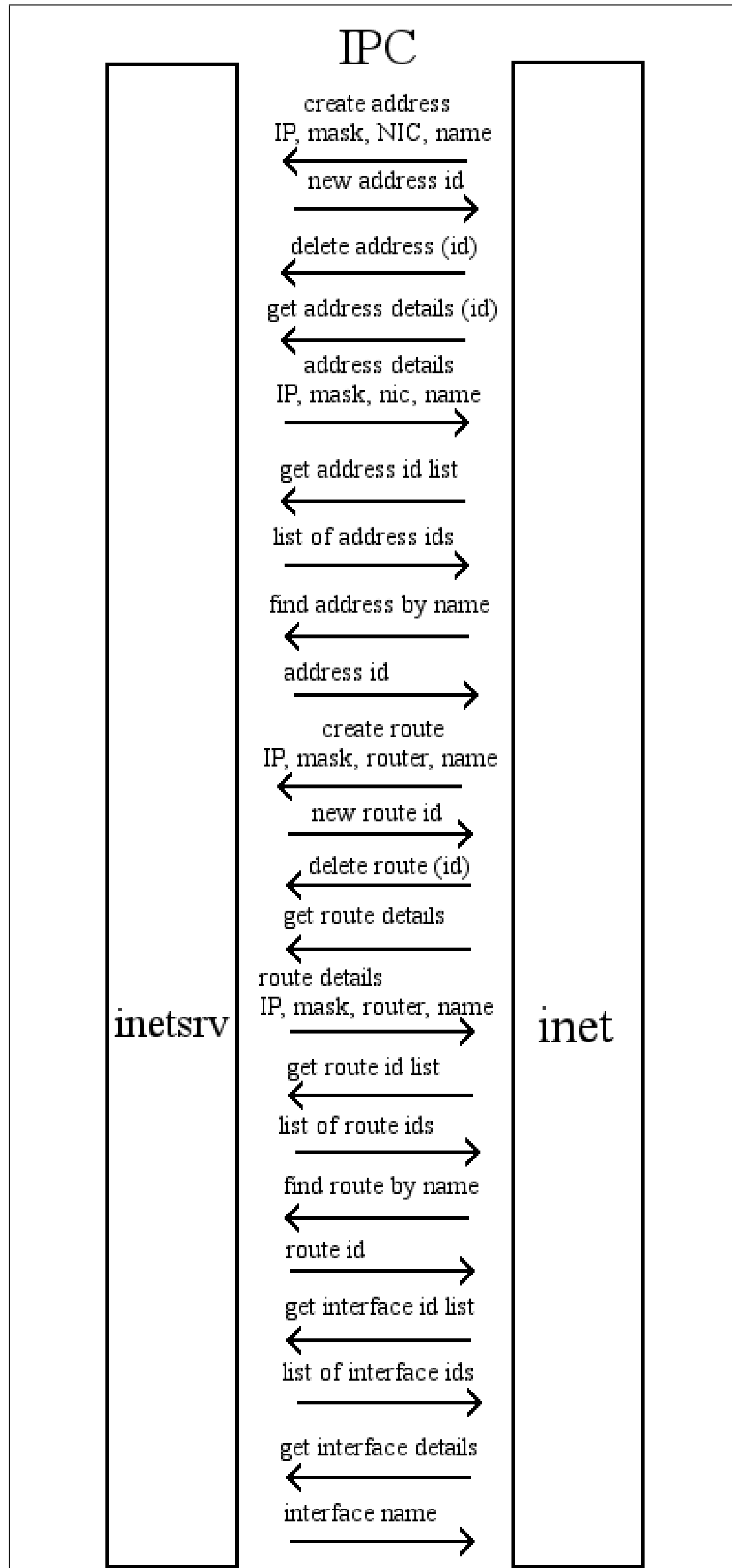
11. IPC between `inetsrv` and transport protocol servers - `inet` port



Next IPC port (`inetcfg`) is used by administration utilities - such as `inet` in `uspace/apps/inet`. This port has the largest number of IPC methods for getting and setting network configuration entries. It is possible to assign an IP address with a network mask to any interface - `inetsrv` returns identifier of just created address (or more precisely network). The administration utility can also get list of all address identifiers or to find an address identifier by its name. When it knows an address identifier, it can delete the address or get its details.

Similar manipulation is allowed with rows in routing table. Administration utility can create a route (network IP address, network mask and router IP address), delete it, get its details, find it by name or get list of all routes. Finally, there is a possibility to get list of all network interface identifiers or details of any network interface (read-only).

12. **IPC between `inetsrv` and administration utilities - `inetcfg` port**



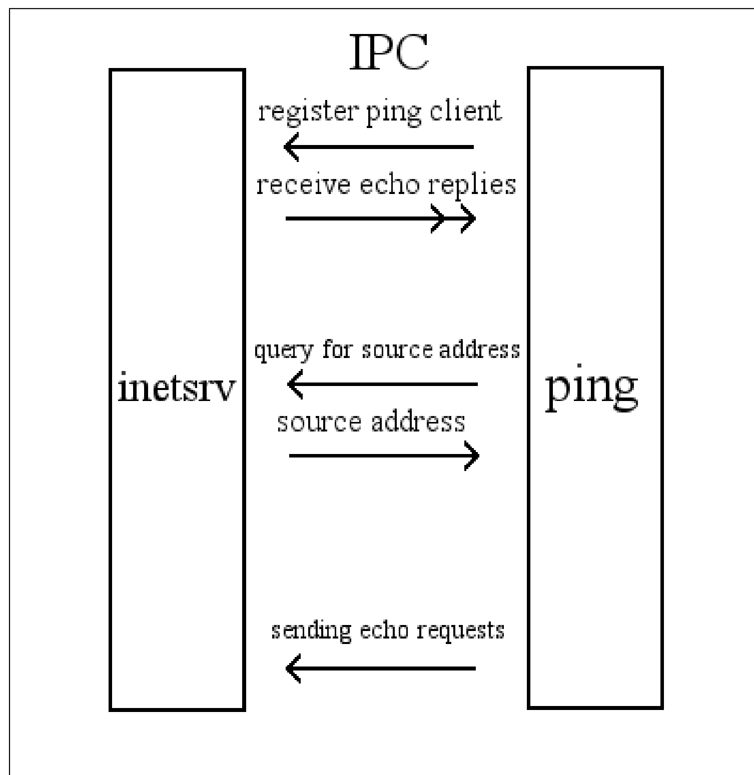
`Inetsrv` implements partially also ICMP protocol. So far, it supports only pinging - *Echo request* and *Echo reply* messages. It automatically replies to

each received *Echo request* message with a correct *Echo reply* message. `Inetsrv`, respectively HelenOS as a whole, does not support classical BSD raw sockets for sending ICMP *Echo request* messages and receiving *Echo reply* messages.

Application pinging to an IP address connects over IPC to that port. `Inetsrv` automatically assigns it an ICMP echo *identifier*. This *identifier* is copied from a static, synchronized and automatically incremented variable to assure always-unique identifier. `Inetsrv` uses assigned *identifier* for all *Echo request* messages sent by that application. Contrarily to BSD raw sockets, pinging application in HelenOS cannot choose its own *identifier*. `Inetping` interface even provides no means how to get the assigned *identifier*. Pinging application can choose only the *sequence number*, IP addresses and packet payload.

When received an ICMP *Echo reply* message, `inetsrv` extracts its *identifier* and looks for it to the list of applications connected to `inetping` port. If it finds an application with the correct ICMP echo *identifier*, it extracts the payload, both IP addresses, *sequence number* and sends the message over IPC to that application. If the *identifier* is not found, the reply is discarded and ignored.

13. IPC between `inetsrv` and pinging applications - `inetping` port



2.3.5 HelenOS transport-layer servers

`Tcp` and `udp` servers both implement equally-named protocols. `Tcp` server is probably the most complicated part of HelenOS networking stack because of TCP

protocol nature. Deep description of its functionality non-related to network-layer protocol (it includes primarily reliability provision mechanisms, handshaking, connection states and congestion control) is not necessary for purposes of this thesis.

`Udp` server holds a list of associations. It is a list of all sockets. Association is uniquely defined by four items: local IP address, local port, foreign IP address and foreign port. Each couple of associations must differ at least in one of these values or the local port must be set to zero.

User application can create a socket over IPC and bind it to a free port. When asked for socket creation, `udp` server creates new association with zero local port (it means unbound socket ignored during packet receiving). When a user application calls *bind* over IPC, `udp` server sets local IP address and local port to requested values. User application can pass over IPC also a *sendto* command. If the socket is even unbound, `udp` server binds it automatically to a free port (looks for an unprivileged port which is even not set as local in any of the associations), creates packet header (including checksum calculation with the pseudoheader) and sends it with both IP addresses over IPC to `inetsrv`.

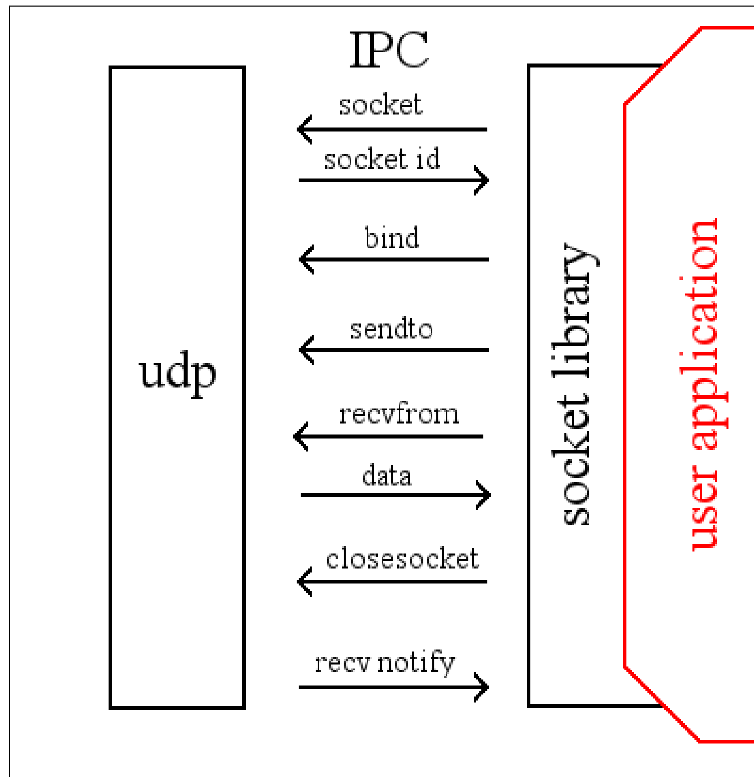
On *closesocket* call, `udp` server just deletes the association and truncates its message-queue. The most complicated command is *recvfrom*. If the message-queue of requested association is not empty, `udp` server just pops its first message and passes it over IPC together with foreign port and foreign address to the application. Otherwise, it orders the application to wait for data until it is notified with a *RECEIVED* message to repeat the request.

When `udp` server receives from `inetsrv` by IPC a UDP packet with source and destination IP, it looks for a matching association. Matching association must have:

1. Local IP address set to `INADDR_ANY` or to destination IP address of the packet
2. Local port set to destination UDP port
3. Foreign IP address set to `INADDR_ANY` or to source IP address of the packet
4. Foreign port set to zero or to source UDP port

If such an association is found, the datagram is inserted into its queue and the application waiting for data on this socket by *recvfrom* is eventually woken by *RECEIVED* IPC message. Otherwise, the packet is discarded and ignored.

14. IPC between udp and user-application using socket library



Tcp server holds a similar structure - list of connections. Each connection is defined by five items: local IP address, local port, foreign IP address, foreign port and listen flag. If the local port is set to zero, its socket is again unbound or invalid (after *RST* packet). Socket with the local port set to zero must be ignored during packet receiving.

If the foreign port is not set to zero, it means the socket successfully called *connect* or it was created by an *accept* call. If the listen flag is set, it means the socket is a listening socket and foreign address with foreign port is ignored. Otherwise the socket is bound, not listening and not connected.

User application can create a connection by *socket* IPC message and eventually bind it. On *bind* IPC message, tcp server just sets its local port and local IP address to wanted values. On *connect* IPC call tcp server firstly checks if the socket is already bound. If it is not, tcp server binds it automatically to a free port. Then it sets foreign IP address and foreign port to wanted values and performs TCP handshake with the target.

On *send* IPC message, tcp server checks if the socket is connected or accepted. If it is not, tcp returns an error. If the socket is connected or accepted, tcp pushes the data into its dispatch queue to be eventually reliably sent and received by the other side. On *recv* IPC call, the server checks again at first validity of the connection. If its receive-queue is not empty, it just pops at most the wanted amount of data and sends it over IPC to the application. Otherwise, it blocks the

application, waits until some data arrive and then it wakes waiting application with a *RECEIVED* notification.

On *listen* IPC message, `tcp` server checks if the socket is bound. *Listen* call handler never binds it automatically contrarily to *connect* handler. Secondly it checks, if the socket is neither connected nor accepted. It means that the foreign IP address is set to `INADDR_ANY` and the foreign port is set to zero. If all checks pass, `tcp` sets the listen flag to true and initializes its accept-queue. Listen flag prevents the socket from connecting it. Eventual *connect* message will be rejected with an error.

Accept is the relatively most complicated call. On *accept* IPC message `tcp` server checks if the socket is listening. If its accept-queue is not empty, it pops the first accepted connection, assigns it a number (socket identifier) and returns it to the calling application. If the accept-queue is empty, `tcp` server orders the application to wait until it will be notified with an *ACCEPTED* IPC message to repeat the call.

When `tcp` server receives from `inetsrv` over IPC a TCP packet with source and destination IP, it looks firstly for an open connection (it means not listening connection with all four values set - both addresses and both ports). It there is a connection, that has:

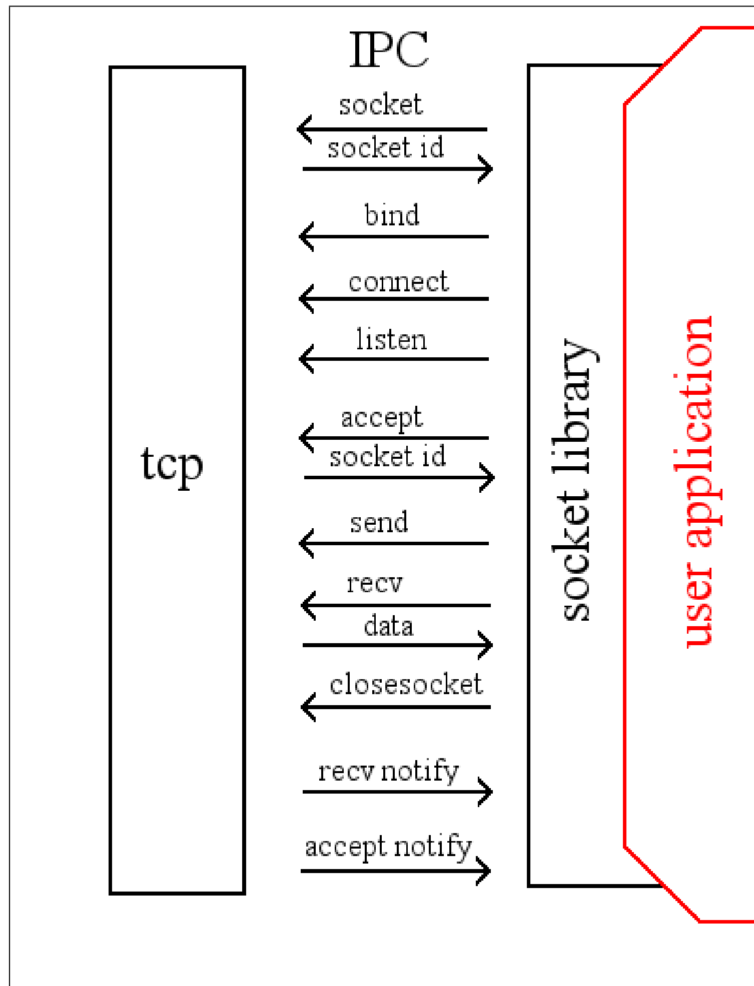
1. Listening flag set to false
2. Local IP address set to `INADDR_ANY` or to destination IP address
3. Local port equal to TCP packet destination port
4. Foreign IP address set to source IP address
5. Foreign port set to TCP packet source port

the packet payload is just pushed to its receive-queue. If there is no such a connection, `tcp` server looks even for a listening socket. It has a sense only if the packet has the *SYN* flag set to true. Such a connection must fulfill these three requirements:

1. Listening flag set to true
2. Local IP address set to `INADDR_ANY` or to destination IP address
3. Local port equal to TCP destination port

If such a connection is found, `tcp` server creates a new connection, pushes it to accept-queue of the original listening connection, sets both its TCP ports and IP addresses and waits for an *SYN/ACK* packet to open it finally. When *ACK* packet finally arrives, `tcp` eventually sends *ACCEPTED* notification to the application waiting in an *accept* call on the listening socket. If nor connected/accepted nor listening connection is found, `tcp` server discards the packet and sends an empty *RST* packet to the sender.

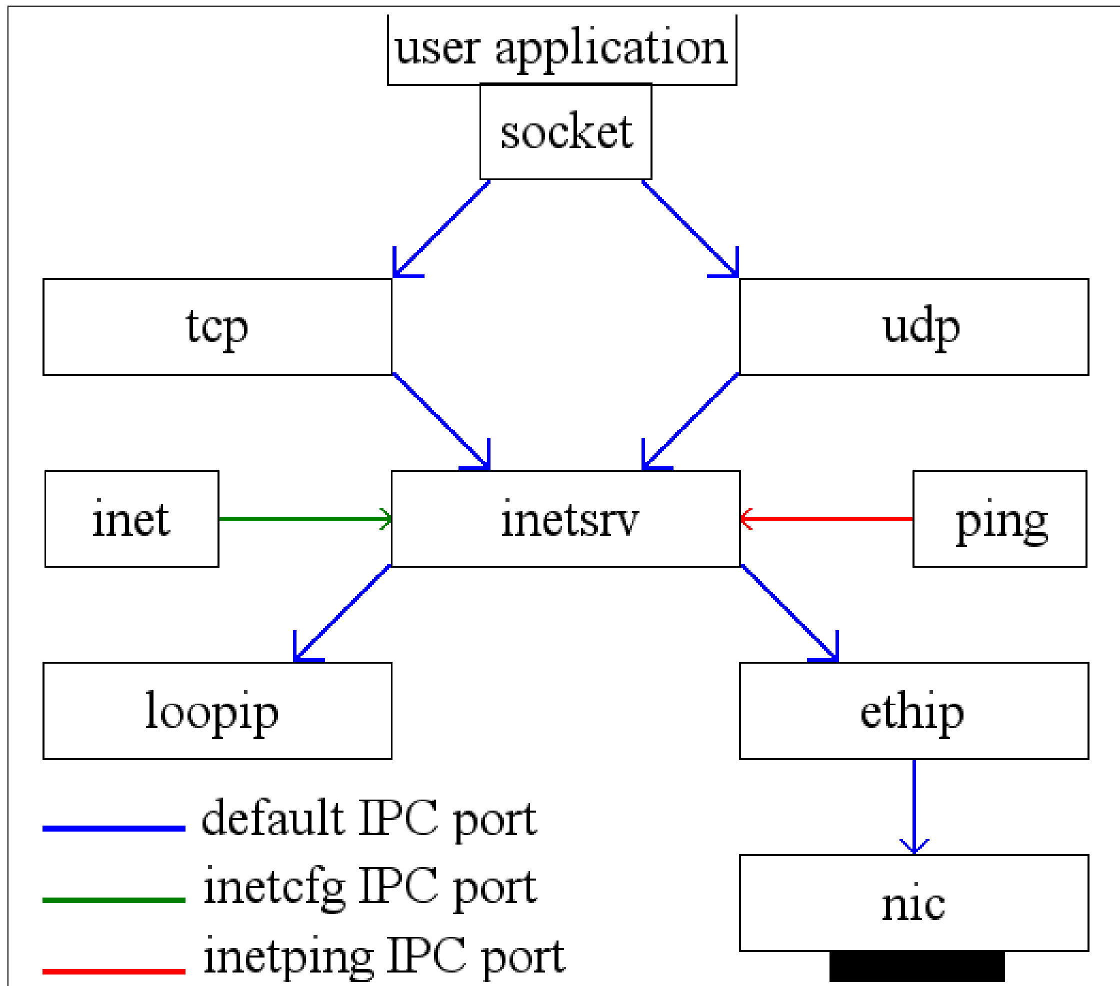
15. IPC between `tcp` and user-application using socket library



2.3.6 HelenOS socket library

Socket library can be linked by any application using BSD socket interface. It just distinguishes sockets administered by `udp` server (*SOCK_DGRAM*) and sockets administered by `tcp` server (*SOCK_STREAM*). Consequently, it connects to correct transport-layer server and encapsulates the partial BSD socket interface into IPC communication with the server.

16. HelenOS networking stack as a whole



3. Analysis

3.1 Strategic decisions

IPv6 is a new protocol. It should be implemented with respect to HelenOS multiserver architecture. IPv4 and IPv6 protocols are similar but they have some substantial differences. This section describes how the new IPv6 implementation would be incorporated into HelenOS networking stack. Coexistence of IPv4 and IPv6 protocols in one networking stack probably requires even some patches of current IPv4 implementation.

3.1.1 Protocol/task question

One of principal high-level decisions for HelenOS IPv6 protocol implementation was the code reuse among existing and eventually some new tasks. It was possible either to implement IPv6 protocol in a completely new task `inet6srv` or to use current task infrastructure and just substantially extend the `inetsrv` task.

Previous HelenOS networking stack, as described in the *Context* chapter, respected some rules. Firstly, each protocol was implemented by a separate task. Secondly, auxiliary protocols such as ARP or ICMP were implemented by the main protocol task. Ethernet task `ethip` implemented also ARP protocol and IP task `inetsrv` implemented ICMP protocol. There was no exact analogy for a new version (mutually incompatible with the previous) of an existing protocol. It was therefore impossible to solve this problem just continuously by an existing rule.

New task addition would lead to better granularity and modularity of the networking stack. It would also automatically solve the problem of IPv6-only and IPv4-only modes. When the user kills `inetsrv` task, he enters automatically IPv6-only operating mode. Similarly, he enters IPv4-only mode when he kills `inet6srv` task. Both `inetsrv` and `inet6srv` tasks running denote dual-stack mode and both killed mean a special mode of completely prohibited networking.

However, that solution equally demanded modifications in the other tasks (mainly `tcp`, `udp`, `ethip` and `loopip`). Their interfaces with `inet6srv` would be similar to their interfaces with `inetsrv` but not equal. There would be differences alone for different address lengths. The networking stack would become less arranged. Transport servers (`tcp` and `udp`) are already the most complicated part of the networking stack. They would need to interact with two different network-layer servers through two different interfaces and to store two different address types.

The main reason for integration of both IP-version addresses and protocols into one existing task `inetsrv` came right from transport-layer servers. Majority of POSIX systems offer sockets of *AF_INET6* family with *IPV6_V6ONLY* option turned off by *setsockopt* (if not turned off by default). Such a socket is dual stack

and it can take both versions of IP protocol. If it is a TCP listening socket bound to *in6addr_any*, it accepts not only IPv6 connections, but also IPv4 connections. If it is a UDP socket, it receives not only IPv6 packets but also IPv4 packets. Dual stack UDP socket can send IPv4 packets by *sendto* using IPv4-mapped IPv6 address. When an IPv4 connection or packet arrives, *recvfrom* and *accept* return IPv4-mapped IPv6 addresses too.

There are some other POSIX OS not supporting dual stack sockets - in particular OpenBSD. It would imply that dual stack sockets are just a voluntary feature of IPv6 implementation. Such an objection is not relevant to HelenOS because HelenOS lacks *select* and *pool* functions. If an OpenBSD TCP or UDP server (e.g. webserver or DNS server) wants to listen on both address families, it just creates two sockets (one *AF_INET* and one *AF_INET6*) and selects on both of them. When a notification comes, it eventually accepts on the right socket and works with created sockets uniformly. HelenOS provides neither *select* nor *pool* functions. Without dual stack sockets it would be impossible to run a dual stack TCP or UDP server. That fact would naturally significantly degrade benefits of HelenOS IPv6 capability.

Therefore, if IPv4 and IPv6 protocols were implemented by different tasks, `tcp` and `udp` servers would have to integrate them by itself relatively chaotically on the transport layer. That is why we decided to integrate network-layer protocols systematically on the network layer in `inetsrv` task. These two network-layer protocols are in fact two versions of just one protocol. Their implementing codes more merge than alternate. Completely different is just their PDU encoding, PDU decoding and MAC address resolution. The other parts are equal or very similar. My solution thus does not add any new tasks. It just significantly extends the `inetsrv` task and partially also the others.

3.1.2 Data structures common to IPv4 and IPv6 addresses and IPC interface from `inetsrv` "up"

Integration of IPv4 and IPv6 in `inetsrv` was intended to keep unique IPC interface between transport-layer servers (`tcp` and `udp`) and `inetsrv` process. `Inetsrv`, `tcp` and `udp` servers hold data structures containing IP addresses - routing table, network interface list, list of UDP associations, list of TCP connections, etc. Forking all those IPC interfaces and data structures into IPv4 versions with IPv4 addresses and IPv6 versions with IPv6 addresses would be very unsuitable and unwanted. It was necessary to unify addresses of both families to be expressed and transferred as one single data type.

First possibility was to create a structure including one byte of address family and a union with both addresses - together 17 bytes (20 bytes with first-item alignment). Such a solution would be possible but it would require indispensable workaround in network masking mechanisms. Network masking mechanism is naturally used in interface network detection and in route selection. The other fork of code would be necessary in socket library translations between *sockaddr_in* and *sockaddr_in6* structures.

Much better solution offered the IPv4 to IPv6 address mapping as described above. There is a trivial one-to-one mapping between IPv4 and IPv4-mapped IPv6 addresses. To translate IPv4 network mask to IPv4-mapped IPv6 network mask (and back) it is necessary to add (or respectively subtract) 96 bits. IPv6 addresses have 128 bits, IPv4 addresses have only 32 bits and 96 is the difference. Common data type is thus simply an IPv6 address of 16 bytes. Native IPv6 addresses will be used unchanged and IPv4 addresses will be mapped into IPv6 by this mechanism. This mechanism prevents ambiguity because the range for IPv4-mapped IPv6 addresses is reserved and no address from it can be ever assigned for a colliding purpose.

This solution has two important implications. All data structures (logically) bearing both IPv4 and IPv6 addresses will represent them uniformly as IPv6 addresses (and some or all of them may be IPv4-mapped). The IPC among single networking stack servers will use also only IPv6 addresses (which may be again IPv4-mapped).

Applications that really must work with IPv4 addresses will need a translation and mapping-detection mechanisms. Among those applications belong administration utilities (such as `uspace/apps/inet`) because the user must be able to set and read both IPv4 and IPv6 addresses in their native formats. Another category is formed by pinging applications (e.g. `uspace/apps/ping`) because they also take raw addresses from users.

Finally, there are non-ignorable differences in BSD socket interface. *AF_INET* sockets use *sockaddr_in* structure for binding and connecting sockets. *AF_INET6* sockets use *sockaddr_in6* structure. Socket library must remember family of the socket by itself and each time check if the socket address was provided in corresponding format.

The transfer of such functionality to transport-layer servers is again unwanted. Those servers have no knowledge of socket families. Transferring local checks to foreign servers yields only higher costs and harder error detection.

Thence it follows that both translation mechanisms between *sockaddr_in* and *sockaddr_in6* address should be implemented already in the socket library. IPC interface should be as simple as possible and it should support just one type of IP address - IPv6. Translation code in transport-layer servers would be even redundant in `udp` and `tcp` because there are no differences. The IPC between socket library and transport-layer servers will thus use only *sockaddr_in6* socket address format.

The only difficulty disallowing complete IPv6 transparency of `tcp` and `udp` servers are *AF_INET* sockets bound to *INADDR_ANY*. Those socket may accept connections (or receive packets) from any IPv4 address and not only from the one mapped address which is moreover invalid. *INADDR_ANY* has a special semantics which is similar to *in6addr_any*. The only difference is that *in6addr_any* matches all and *INADDR_ANY* only IPv4-mapped addresses.

Tcp and udp servers will hence work only with IPv6 addresses. All their IPC interfaces (both to inetsrv and to socket library) will use only IPv6 addresses too. The only two patches to ensure IPv4 compatibility will consist in *INADDR_ANY*-mapped IPv6 address check and in checksum calculation.

3.1.3 NDP position (link-local addresses) and the rest of IPC interface

Interfaces and data structures among network-layer server, transport-layer servers and socket library are practically solved and clear. IPv4 addresses will be completely replaced with IPv6 addresses and eventually mapped. Possible translation to native IPv4 addresses will be performed by "end nodes" of the networking stack. End nodes are socket library, administration utilities and pingers. All components will be necessarily patched to fulfill IPv4 specialties intrinsically incompatible with IPv6 (such as pseudoheader checksum calculations). Much more complicated is the bottom part of HelenOS networking stack. It means from inetsrv server down.

Inetsrv itself must distinguish between native and IPv4-mapped addresses thought they all would be stored uniformly. PDU encoding will be completely different for IPv4 and for IPv6 datagrams. IPv6 datagram fragmentation and reassembling is much less necessary if the outgoing datagrams will obey the minimal MTU of 1280 bytes. All NICs will naturally accept the maximum of 1500 bytes for incoming datagrams.

Probably the main difference between IPv4 and IPv6 networking stack architecture consists in completely different MAC address resolution. IPv4 uses ARP as a part of link-layer Ethernet protocol. IPv6 uses NDP. NDP is a part of network-layer ICMPv6 protocol.

There were two possibilities but none of them was backward compatible with the previous implementation and its principles. Inetsrv was previously unaware of MAC addresses. NDP is a part of IP protocol and therefore it should be implemented by the IP server.

First possibility was to implement NDP protocol in ethip process. Ethip would be aware of all IPv6 addresses assigned to a particular interface. It would recognize and intercept every Ethernet frame carrying NDP packet and process it by itself. Advantages of that approach include IPC interface preservation (only IPv4 to IPv6 changes). Disadvantages include complete breakdown of protocol/process conception together with chaotic and redundant processing and creation of IPv6 headers, ICMPv6 headers and even NDP bodies and options in a task originally processing just Ethernet frames.

Second possibility was to enhance IPC interface between inetsrv and link-layer servers (loopip and ethip) by MAC address passing. It means that ethip returns on demand MAC address of a particular NIC and inetsrv can send a

datagram with a pre-set target MAC address. MAC address resolution of IPv6 datagrams would be moved into `inetsrv` task. Advantages of this approach include mainly protocol/process conception compatibility. Disadvantages include non-trivial IPC interface upgrade needs.

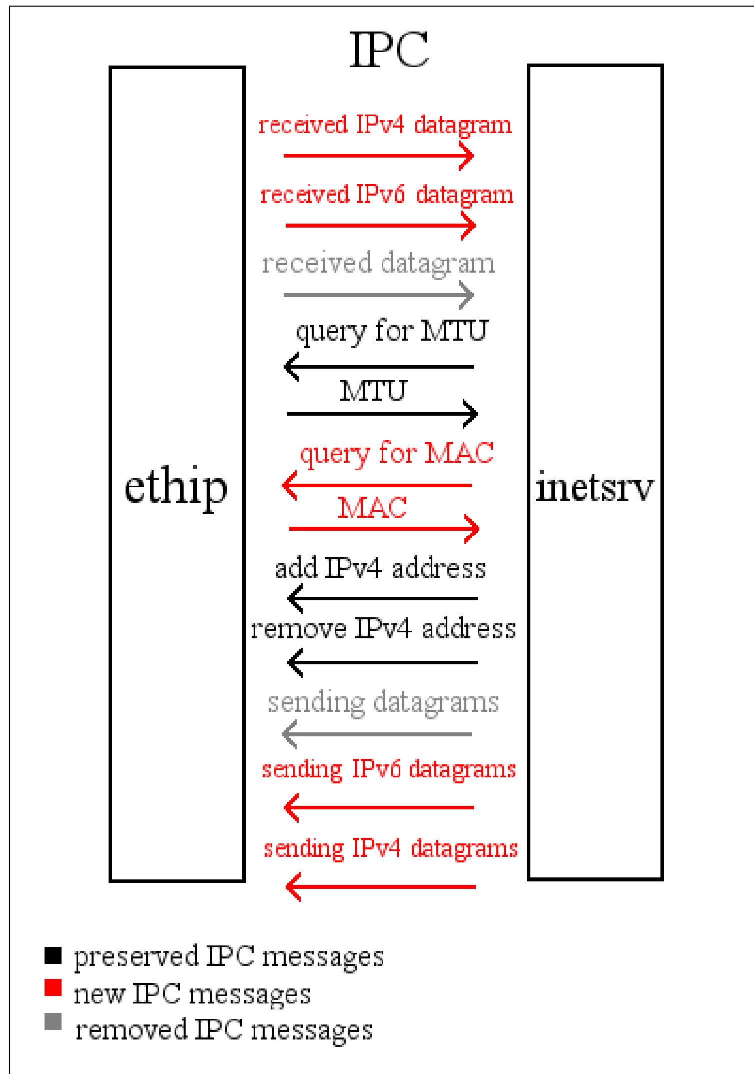
Finally, the second possibility was chosen because of respect to HelenOS multiserver architecture. It is a much more systematic solution though it modifies and complicates interfaces.

New IPC commands are MAC address query and the possibility to send a datagram with already pre-set target MAC address. Whole ICMPv6 including NDP will be processed directly in `inetsrv` task analogically to ICMP. NDP will form a separate module of `inetsrv` with a conception very similar to ARP. Configuration module of `inetsrv` will pass only IPv4 addresses to link-layer servers. `Ethip` and `loopip` stay unaware of IPv6 addresses because ARP does not need them. IPC between `inetsrv` and link-layer servers preserves IPv4 communication (4-bytes addresses) because no IPv6 addresses can go through it. `Ethip` task will distinct IPv4 and IPv6 datagrams by their *ether-type* and send them to `inetsrv` by different IPC commands.

3.2 Interface enhancements

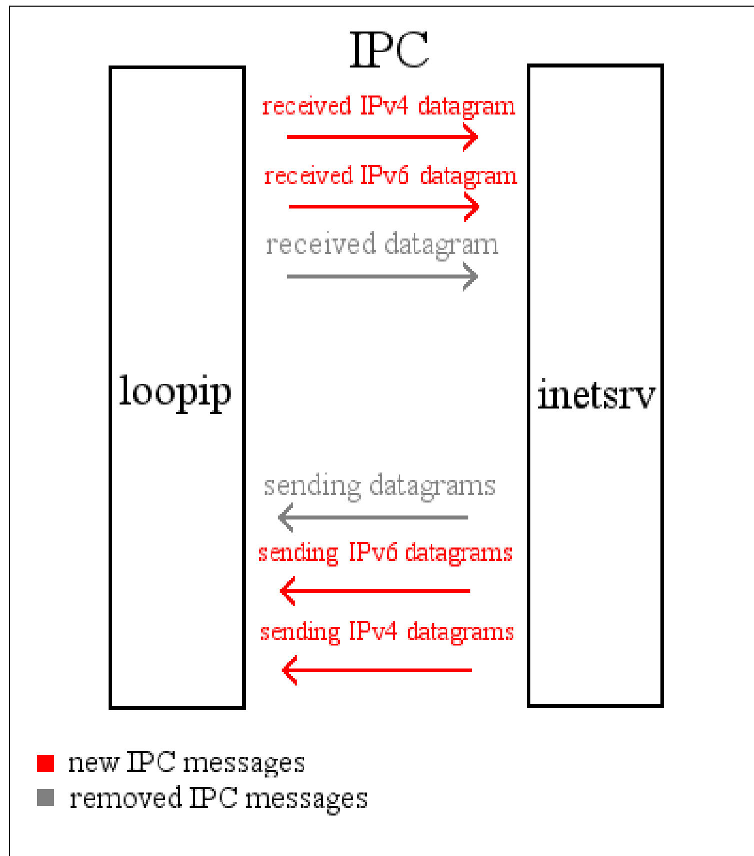
IPC interface between `nic` and `ethip` servers remains untouched. IPC between `ethip` and `inetsrv` servers will be significantly upgraded by adding and removing some IPC message types. `Ethip` will recognize frames encapsulating IPv4 datagrams and IPv6 datagrams by their *ether-type* field. It will send their payloads by different IPC commands to `inetsrv` to be sooner distinguished. `Inetsrv` will reciprocally use two IPC commands for sending IPv4 and IPv6 datagrams. What is more, those two IPC messages will have very different parameters. The first sends source IPv4 address, target IPv4 address and serialized IPv4 datagram. The second sends just target MAC address and serialized IPv6 datagram. IPv6 MAC address resolution is performed already by `inetsrv`. The last IPC interface upgrade between `ethip` and `inetsrv` is the own MAC address query which is very similar to MTU query.

17. Changes in IPC between `ethip` and `inetsrv`



Separate problem is the `loopip` process. Its IPC interface with `inetsrv` should be similar to `ethip`/`inetsrv` interface but loopback has no MAC addresses. The patch is simple. `Loopip` will always return zero MAC address of loopback (MAC address with all bits set to zero) and `inetsrv` consequently turns off NDP for such interface. It sets target MAC address of every outgoing datagram (through that interface) to zero too. `Loopip` just pops that zero MAC address and flags the datagram as IPv6 to be correctly "received" and returned back as IPv6 to `inetsrv`. It similarly stamps IPv4 datagrams to emulate the *ether-type* mechanism in `ethip`.

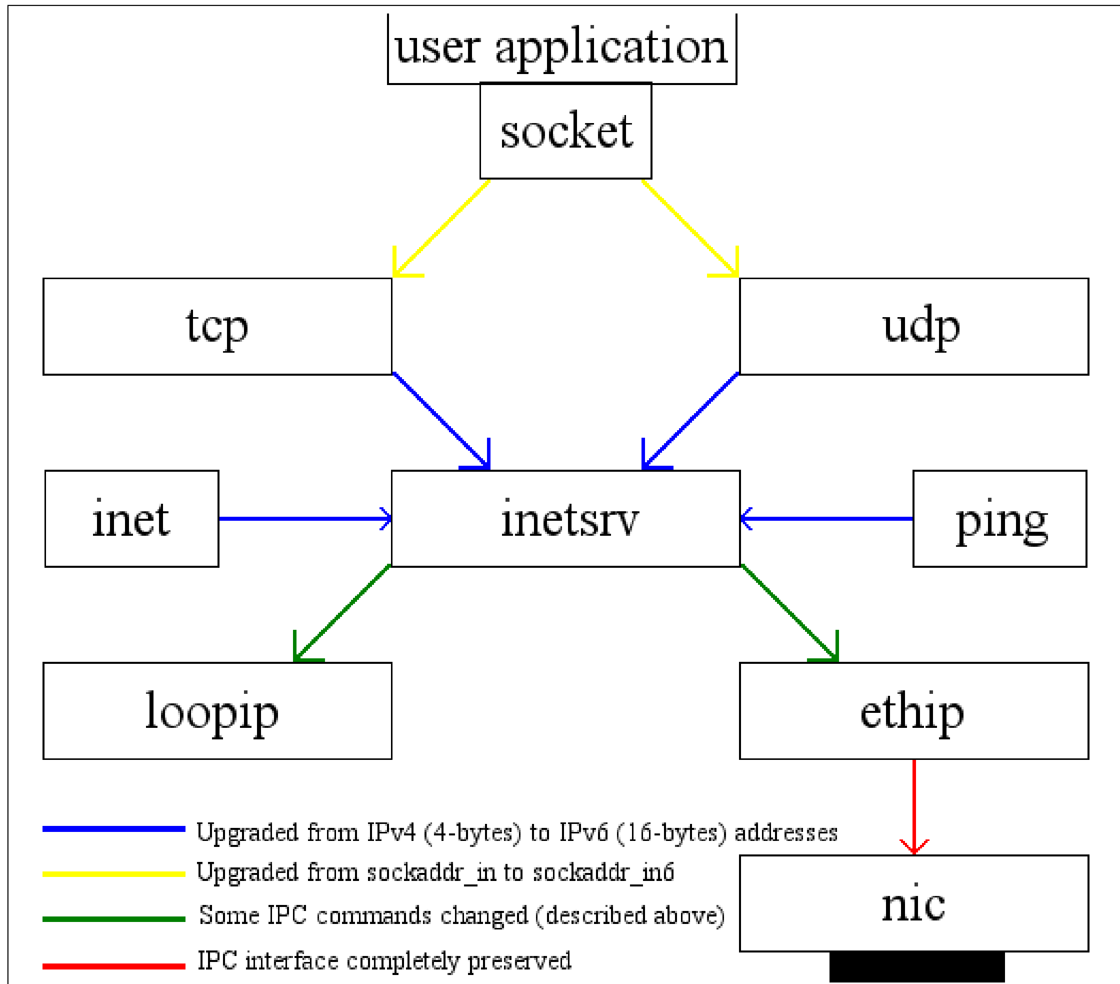
18. Changes in IPC between `loopip` and `inetsrv`



IPC between `inetsrv` and transport-layer servers will be upgraded very simply. All IPv4 address (4-byte values) appearances will be replaced by IPv6 addresses (16-byte values). IPC larger data block transfers will be necessary for IPv6 address transporting. Basic IPC messaging supports at most 5 integer values which is not enough for two IPv6 addresses on 32-bit computers.

Interface between socket library and transport-layer servers will be upgraded also simply. IPC messages *bind*, *connect*, *sendto*, *accept* and *recvfrom* will no more use *sockaddr_in* structure (16-byte value) but exclusively *sockaddr_in6* structure (28-byte value).

19. **Structure of HelenOS networking stack interface upgrade**



3.3 Code upgrades

This section contains already the low-level specification of required code changes. It should be as detailed as possible in the *Analysis* phase.

3.3.1 Changes in shared libraries (others than socket library)

Shared libraries contain features demanded by multiple applications (OS servers, utilities, user-applications). With IPv6 capability of HelenOS come new needs of conversion functions, appropriate data types and constants.

It was necessary to define standard structures *sockaddr_in6*, *in6_addr* and constants *in6addr_any* and *in6addr_loopback*. Special constant *in4addr_any*, which is in fact *INADDR_ANY*-mapped to IPv6, is useful for transport-layer servers and for the socket library. *In6addr_multicast_all_nodes* represents `ff02::1` address, universal link-local destination. All nodes are obliged to accept datagrams destined to there. Finally, it was necessary to define newly useful standard macro

values such as *IP_PROTO_ICMPV6*, *ETH_P_IPV6*, *ICMPV6_ECHO_REQUEST*, *ICMPV6_NEIGHBOR_SOLICITATION*, etc.

Among standard conversion functions, it was necessary to implement correct *inet_pton* and *inet_ntop* functions. Their implementation was casuistically switched to IPv4 and IPv6 version (by family parameter). These functions have the same abilities as their correlates in GNU C library. They accept IPv4 abbreviated notation (e.g. `127.1` is `127.0.0.1`) and both accept and return IPv6 abbreviated notation (e.g. `::1` is `0000:0000:0000:0000:0000:0000:0000:0001`).

Auxiliary function *is_mapped* has one parameter (IPv6 address). It returns true if the address is IPv4-mapped and false if it is not. Function *ipv4_to_ipv6* converts IPv4 address to IPv4-mapped IPv6 address. *Ipv6_to_ipv4* function is reversion of *ipv4_to_ipv6* converting it eventually back. *In6addr_equal* compares two IPv6 addresses for equality because C-language provides no means how to compare two 16-byte arrays directly.

Inner libraries for IPC communication from transport-protocol servers with `inetsrv` (*inet.h*), for pingers with `inetsrv` (*inetping.h*), for administration utilities with `inetsrv` (*inetcfg.h*), for `inetsrv` with link-layer servers (*iplink.h*) and for link-layer servers with `inetsrv` (*iplink_srv.h*) were upgraded just to use new IPC interfaces as described in previous chapter.

3.3.2 Changes in `ethip`

Ethernet protocol implementation must newly accept MAC address multicasts. At least IPv6 multicast (MAC addresses beginning with `33:33:`) are necessary. Except that, `ethip` must only adapt new IPC interface towards `inetsrv`.

`Ethip` must newly recognize *ETH_P_IPV6* frames and send their payload to `inetsrv` by *IPLINK_EV_RECV_IPV6* IPC command. It will continue in self-processing *ETH_P_ARP* frames and in forwarding *ETH_P_IP* frames by *IPLINK_EV_RECV_IPV4* IPC command to `inetsrv`.

Implementation of new IPC message *IPLINK_GET_HWADDR* will not be complicated. `Ethip` just reads MAC address from interface's `ethip_nic_t` structure, splits it into two 4-byte numbers (first two bytes to one integer and the rest to another one) and return these two integers by IPC response to `inetsrv`.

IPLINK_SEND_IPV4 will be implemented the same way as previous *IPLINK_SEND* IPC message. *IPLINK_SEND_IPV6* implementation just reads two integers (parameters of this IPC-message type) and joins them into target MAC address value. Source MAC address will be read from `ethip_nic_t` structure and the frame is to be immediately sent without any ARP resolution and other complications.

Finally, `ethip` will be affected by moving MAC address headers and utilities to a shared library because the same functions will be used by NDP stack in `inetsrv` server. This appertains to `mac48_addr_t` data type, *MAC48_BROADCAST* macro and to `mac48_encode` and `mac48_decode` functions. Those functions will be declared in `net/ether.h` file and moved to `uspace/lib/c/generic/net/ether.c` library.

`Ethip` supports `--ipv4-only` and `--ipv6-only` command-line arguments. With the first one, `ethip` enters IPv4-only mode ignoring all Ethernet frames with IPv6 datagrams and blocking all *IPLINK_SEND_IPV6* IPC messages from `inetsrv`. With the second one, it enters IPv6-only mode ignoring all Ethernet frames with IPv4 or ARP datagrams and blocking all *IPLINK_SEND_IPV4* IPC messages.

3.3.3 Changes in `loopip`

`Loopip` only needs (analogically as `ethip`) to ensure distinction between IPv4 and IPv6 datagrams. The receiving interface is strongly-typed. IPv4 datagrams arrive by *IPLINK_SEND_IPV4* message. IPv6 datagrams arrive by *IPLINK_SEND_IPV6* message. `Loopip` thus suffices with a new member in loopback-queue item storing IP version of the datagram. Value of that member will be checked after consuming the item from loopback queue. Matching payload will be returned over IPC by *IPLINK_EV_RECV_IPV4* or by *IPLINK_EV_RECV_IPV6* message back to `inetsrv`.

3.3.4 Changes in `inetsrv`

Address representation

All data structures representing IPv4 addresses will be replaced with unified IPv6 representation - structure *in6_addr*. Data type expressing an IPv4 address with its netmask (used in *addrobj* module to express IP address assigned to a particular interface and in *sroute* module as a route destination) will be replaced by IPv6 address with netmask. New couple of conversion function will be implemented. Function converting IPv4 network to the new data type converts IPv4 address to IPv4-mapped IPv6 address (using `ipv4_to_ipv6` function) and increases the netmask by 96 bits. Backward conversion calls `ipv6_to_ipv4` to the address and it subtracts 96 bits from the netmask.

Netmasking mechanism will be completely rewritten because of new IP address representation but its principle remains the same. Checking function will have just two parameters. First parameter contains IPv6 address with netmask to express network range. Second parameter contains IPv6 address to be checked whether it belongs to the network range or not. The function just generates netmask as 16-byte array, masks both addresses and then checks their equality.

`Inetsrv` must also anew convert IPv4-mapped IPv6 addresses added by `inetcfg` IPC port to IPv4 before they are forwarded to `ethip`. The IPC interface between `inetsrv` and `ethip` preserves IPv4 addresses contrarily to `inetcfg`. Native IPv6

addresses added by `inetcfg` will not be naturally passed to `ethip` because `ethip` does not need them as explained above.

New destination IPv6 addresses

`Inetsrv` will accept IP datagrams destined to multiple addresses. Classical unicast addressing known from IPv4 remains. Solicited-node multicast addressing accedes because Neighbor solicitation messages are necessary for MAC address resolution and they are destined only that way. Universal link-local multicast addresses (such as `ff02::1`) will be partially supported but their implementation may not be complete. Neither IPv4 implementation does accept IP datagrams destined to broadcast IPv4 addresses. There must be defined a new constant variable solicited-node multicast mask (`ff02::1:ff00::/104`) for netmasking IPv6 destination addresses. New couple of functions converting target IPv6 address to IPv6 multicast solicited destination (*ndp_solicited_node_multicast_address*) and to MAC target of its encapsulating Ethernet frame (*ndp_solicited_node_mac_address*) will be introduced.

ICMPv6 protocol implementation

New protocol ICMPv6 will be implemented directly by `inetsrv` task in a new file *icmpv6.c*. `Inet_recv_dgram_local` function will not check only ICMP. It will check also ICMPv6 value before searching in available transport protocols. It will process ICMPv6 packets by itself forwarding them to `icmpv6_recv` function.

ICMPv6 implementation will be independent from current ICMP implementation. Their architectures will not have many substantial differences. The checksum calculation is different. ICMP does not use pseudoheaders while ICMPv6 does. Matching ICMP and ICMPv6 message types use different numbers. Standard ICMPv6 headers and constants are defined in *icmpv6_std.h* file. Both ICMP a ICMPv6 will be connected to *Inetping* module. *Inetping* module resolves what IP protocol version is going to be used and chooses matching control protocol version for any outgoing *Echo request* message. This distinction is performed naturally by *is_mapped* function. `Inetping` IPC interface consisting in automatic assignment of ICMP(v6) echo *identifier* will remain.

ICMPv6 stack will support four message types - *Echo request*, *Echo reply*, *Neighbor solicitation* and *Neighbor advertisement*. *Neighbor solicitation* and *Neighbor advertisement* messages will be redirected to NDP module implemented in *ndp.c* file. `Inetsrv` automatically replies every received ICMPv6 *Echo request* message. ICMPv6 *Echo reply* message will be passed to *Inetping* module and then perhaps to a pinging application with the matching ICMP(v6) echo *identifier*.

MAC address resolution

New IPC interface demands `inet_link_t` structure extension by MAC address of the interface (48 lower bits of 64-bit number). MAC address value will be received from the link-layer server by the IPC call in two message arguments

during NIC opening in `inet_link_check_new`. Since then, `inetsrv` knows MAC address of every interface and NDP module can successfully read it.

Neighbor solicitation and *Neighbor advertisement* ICMPv6 messages will be passed from ICMPv6 module to NDP module. NDP module is in turn similar to ARP implementation in `ethip`. The module holds its main data structure - unsorted linked-list of pairs containing IPv6 addresses and matching MAC addresses.

NDP module extracts message sender MAC and IPv6 address, ICMPv6 type and destination IPv6 address. Then it always saves sender pair of MAC and IPv6 address to the list. It must save it even if it is a *Neighbor solicitation* message to avoid endless solicitation looping. If it is a *Neighbor solicitation* message destined to one of IPv6 addresses assigned to incoming NIC, it replies with correct *Neighbor advertisement* message. NDP PDU encoding and decoding routines will be implemented in *pdu.c* file together with IP datagram PDU routines. These routines are technically relatively complicated. Each *Neighbor advertisement* message will have set *OVERRIDE* and *SOLICITED* flags because HelenOS will send only solicited advertisements. *Neighbor solicitation* messages use naturally no flags.

For principally each IPv6 datagram before it is send (passed to `ethip` by IPC) `inetsrv` must resolve its target MAC address. There are two exceptions to this rule. First exception is formed by NDP messages as described later. Second exception is represented by datagrams that will be send from NIC with zero MAC address (it means loopback). NDP-message datagrams will have their target MAC address pre-set by NDP module. Loopback datagrams target MAC address will be set constantly to zero because `loopip` anyway ignores it.

Either already the destination IP (if the destination is directly in the network of outgoing address) or the first-hop router will be used as target IPv6 address. *Inetsrv* looks to IPv6-MAC pair list for the target IPv6 address. If the lookup is successful, *inetsrv* just assigns found target MAC address and sends the datagram. Otherwise, it builds correct *Neighbor solicitation* message (including correct multicast IPv6 target and multicast MAC target), sends it and waits until correct *Neighbor advertisement* message comes. If the advertisement does not arrive in one second, *inetsrv* gives up waiting and discards the outgoing datagram.

Neighbor advertisement and *Neighbor solicitation* messages never use recursively NDP MAC address resolution. They bypass it using `inet_link_send_dgram` function directly. Target MAC addresses of both message types are defined by ICMPv6 protocol specification. They are derived from destination IPv6 address or equal to solicitation source MAC address. No MAC address resolution is needed to send them. NDP messages neither use routing mechanisms because their target is always in a network of the outgoing interface. NDP will thus remember only the outgoing NIC and use it for related NDP messages instead of their routing. Outgoing NIC of a *Neighbor advertisement* message is the NIC what the

Neighbor solicitation message came from. Outgoing NIC of a *Neighbor solicitation* message in the NIC where the datagram with unknown target MAC address will be sent.

NDP module will use *ether.h* library for MAC address manipulation and storage. It was set apart from `ethip` task to a shared library. IPv4 datagrams will continue in ARP resolution implemented in `ethip`. `Inetsrv` only passes their source and target IPv4 addresses over IPC to `ethip` and `ethip` performs the ARP lookup.

Other changes of `inetsrv`

PDU implementation will be completely forked to IPv4 and IPv6 versions because they have not much in common. IPv4 version remains as it is and IPv6 implementation will be added. It will be simpler than IPv4 fork because of missing internet-layer checksum calculation and simpler fragmentation rules. IPv6 PDU decoding routine firstly decodes IPv6 datagram header ignoring deprecated header fields of IPv4 datagrams such as *type of service*, *checksum*, *identifier*, *offset* and fragmentation *flags*. Only if the *next header* is *fragmentation*, fragmentation parameters will be eventually assigned. PDU module will include NDP message encoding and decoding. Standard header data-types (IPv6 datagram header, NDP message skeleton and NDP option format) will be defined in *inetsrv.h* and in *ndp.h* files.

Automatic setting of constant IPv4 addresses will be erased with no replacement. Contrariwise, IPv6 link-local address automatic assignment will be introduced. Link-local address is derived from the MAC address by `mac_to_link_local` function. If MAC address of a NIC will be zero, then `::1/128` (loopback definition) link-local address will be automatically assigned. Loopback will be automatically assigned also with wide-known `127.0.0.1/8` IPv4 address.

`Inetsrv` launched with `--ipv4-only` argument forbids passing IPv6 datagrams over IPC to link-layer servers and receiving them. It disables *IPLINK_EV_RECV_IPV6* and *IPLINK_SEND_IPV6* IPC messages. Thus, it completely blocks IPv6 communication. `--ipv6-only` argument does the same with IPv4 datagrams. It disables *IPLINK_EV_RECV_IPV4* and *IPLINK_SEND_IPV4* IPC messages. Using one of those arguments HelenOS networking stack enters IPv4-only or IPv6-only mode. `Ethip` arguments `--ipv4-only` and `--ipv6-only` are just an opportune optimization.

3.3.5 Changes in transport-layer servers

Transport-layer servers need only three minor updates. The first difference is different format of associations and connections. IPv4 addresses will be replaced with IPv6 addresses and those addresses will be directly used in both ways of IPC communication. The second update is forked pseudoheader checksum calculation because format of TCP/UDP over IPv4 and TCP/UDP over IPv6 checksum pseudoheader is different as described in *Context* chapter. The last change is

matching of local address to destination IP address and foreign address to source IP address. Universal value will be changed from *INADDR_ANY* to *in6addr_any*. *In4addr_any* (as defined above) will additionally match even all IPv4-mapped addresses but not the others.

3.3.6 Changes in socket library

Socket library must newly ensure and implement socket address conversions. It must also check compatibility between socket family and socket address family.

Each socket structure will contain new item called *family* storing to what family the socket belongs. So far, *AF_INET* and *AF_INET6* families are supported. The item will be assigned by *socket* function from its *family* argument or by *accept* function copying family of the parent listening socket.

Inside *connect*, *bind* and *sendto* functions, socket library must check and let go only two alternatives. First alternative is that both socket and socket address families are *AF_INET* and the socket address length is exactly 16 (size of *sockaddr_in* structure). Second alternative is that both socket and socket address families are *AF_INET6* and the socket length is exactly 28 (size of *sockaddr_in6* structure). Any other combination is invalid and affected function must hence immediately return an error.

If the socket family was *AF_INET*, socket library must create a new structure *sockaddr_in6*. It copies the port number directly from provided *sockaddr_in* structure. New IPv6 address will be obtained calling `ipv4_to_ipv6` on *sin_addr* IPv4 address. Resulting *sockaddr_in6* structure will be send through IPC to corresponding transport-layer server. If the socket family was *AF_INET6*, socket library may omit this workaround and immediately send provided *sockaddr_in6* structure by IPC to the transport-layer server.

A little more complicated are *accept* and *recvfrom* calls. Such functions let pass again only two variants. Either both socket and socket address families are *AF_INET* and the socket address length is at least 16, or both socket and socket address families are *AF_INET6* and the socket address length is at least 28. Any other combination of parameters will not pass through.

If that check passes, socket library receives always a *sockaddr_in6* structure by IPC from transport-layer server. If the socket family was *AF_INET6*, it just fills prepared buffer by that address, the length by 28 (size of *sockaddr_in6* structure) and successfully returns.

If the socket family was *AF_INET*, it must at first convert received *sockaddr_in6* structure to *sockaddr_in*. Socket library converts received IPv6 address by `ipv6_to_ipv4` and assigns the result to *sin_addr* item of the prepared buffer. It copies *sin6_port* of received structure to *sin_port*. *AF_INET* constant is assigned to *sin_family* item and 16 (size of *sockaddr_in* structure) to the socket address length.

3.3.7 Changes in utilities

New utility is to be introduced - `ping6`. Its functionality is analogical to `ping`. It works only with IPv6 addresses while `ping` works only with IPv4 addresses. Naturally, it connects over `inetping` IPC port to `inetsrv` and converts textual representation of IPv6 address to *in6_addr_t* structure.

Old `ping` utility admits only one change. It must convert IPv4 addresses to IPv4-mapped IPv6 addresses because `inetping` IPC interface counts only with IPv6 addresses.

`Inet` utility will integrate both IP protocol versions. It will introduce new function for distinction between IPv4 and IPv6 address textual representation to know with what address family to call *inet_pton*. `Inet` will recognize also received IP addresses from `inetcfg` IPC by *is_mapped* function. IPv4-mapped IPv6 address it converts firstly to native IPv4 and prints the result by *inet_ntop*. Native IPv6 address it prints directly with *inet_ntop*.

4. Implementation

4.1 Step-by-step development and continuous testing

Overall implementation strategy consisted in wide salami-tactics usage and in modular development. It was necessary to define a reasonable subset of testing impositions and charges (testing set). This set had to be correctly working under previous IPv4 implementation. New implementation continued gradually. All those tests were performed after each partial improvement. If something had broken in the foregone development phase, it should not be difficult to find the bug after the phase in the last diff. In the worst case, it was possible to rollback the foregone phase.

First strategic step was to prepare networking stack infrastructure to use IPv6 addresses in all data structures and in all IPC interfaces. Second step was to implement distinction mechanisms (between native IPv6 and IPv4-mapped IPv6 addresses) and to fork implementations of divergent modules. Only the last step was working with the real IPv6 (respectively dual stack) traffic. IPv4 communication should be working correctly during whole implementation process.

4.2 Debugging methods

Operating systems are generally hard to debug. Contrarily, the development of separate userland tasks is much easier than kernel development thanks to all security checks of user-application programming. HelenOS multiserver architecture provides IPC interfaces. IPC interfaces are very useful as inner interfaces to perform modular development. Tasks can be considered as modules and their inputs and outputs are defined by their IPC. The only swinging charge is the precise definition of all affected IPC interfaces. This definition must be unambiguous. It must cover exactly all necessary features and capabilities. After that definition, the rest is a pure example of modular development.

HelenOS lacks advanced testing utilities such as Valgrind. The only useful general-purpose debugging techniques are logging and control outputs. Networking applications can be moreover debugged by network sniffing tools listening on TAP interface such as Tcpdump or Wireshark. These tools are able to check integrity and correctness of outgoing frames, datagrams and packets. We used this possibility very often when introducing new features of the networking stack. Specific instrument is debugging directly by Linux kernel networking stack. Linux kernel detects some parameter errors (e.g., *payload length* of IPv6 datagram with M fragment flag set must be divisible by eight) and eventually replies it with a correct ICMPv6 error message. Wireshark cannot detect directly those parameter errors but it can capture the ICMPv6 error message, sent by Linux kernel, on its way back to HelenOS.

4.3 Testing set

Testing set is intended to be perfectly working under the previous implementation to use the salami-tactics. This set must also cover all non-trivial features of networking stack. Choice of such features and preparation of such tests is thus very important. Outer interface of HelenOS networking stack will be explored to get list of features. Each found feature can have moreover many procedurally different variants and the testing set must cover all of them.

4.3.1 Testing set determination

Outer interface of previous HelenOS networking stack provides these features:

1. Socket interface (subset for UDP and subset for TCP)
2. `Ping` user interface
3. `Inet` (administration utility) user interface
4. Ethernet frames encoding and decoding
5. ARP datagrams encoding and decoding
6. Automatic replies on ARP requests
7. ARP translations
8. IPv4 datagrams encoding and decoding
9. ICMP packets encoding and decoding
10. Automatic replies on *Echo request* messages
11. DNS translations
12. Sending and receiving IPv4 datagrams over loopback

After IPv6 implementation this interface will be extended by:

13. *AF_INET6* sockets, *sockaddr_in6* addresses, mapping IPv4 to IPv6 anywhere possible
14. `Ping6` user interface
15. `Inet` (administration utility) user interface - complete IPv6 address support
16. IPv6 datagrams encoding and decoding
17. ICMPv6 packets encoding and decoding
18. Automatic replies on ICMPv6 *Echo request* messages
19. Automatic replies on ICMPv6 *Neighbor solicitation* messages
20. NDP translations
21. DNS IPv6 servers and *AAAA* records
22. Sending and receiving IPv6 datagrams over loopback

Inet user interface can be easily covered with *create*, *delete*, *add-sr* and *del-sr* commands. Ping interface has three procedurally different variants. First variant of pinging is pinging to loopback (covering also points 8, 10 and 12). Second variant is pinging to a computer in the same Ethernet network (covering also points 4, 5, 7, 8 and 9). The third variant is pinging to a computer in a foreign network covering the same points but checking also the routing mechanism. Point 10 can be covered by pinging to HelenOS from another computer over Ethernet (covering also points 4, 5, 6, 7, 8 and 9). DNS translations are covered by dnscfg *set-ns* and dnsres commands.

The most complicated part of outer interface is the socket interface - point one. UDP socket implementation in HelenOS library supports *socket*, *bind*, *sendto*, *recvfrom* and *closesocket* functions. TCP socket implementation supports *socket*, *bind*, *listen*, *accept*, *connect*, *recv*, *send* and *closesocket* functions. All those functions must be included into tests. At first only *AF_INET* sockets and addresses must be working, later also *AF_INET6* sockets with *AF_INET6* native and *AF_INET6* IPv4-mapped addresses. Covering of this point is described in the next subsection.

Points with numbers 13 and following are to be covered eventually after appropriate features will be implemented. Their covering is described in subsection *Final testing set*.

4.3.2 Antecedent bug fixes in previous IPv4 implementation

Covering supported part of BSD socket interface with appropriate tests was relatively complicated. Testing tools were missing or broken. In uspace/apps directory were some network testing tools (netecho, nettest1, nettest2 and nettest3) covering all supported functions of BSD socket interface. None of them was correctly working in the mainline revision 1841 we came from.

Those tools did not obey HelenOS coding-style and their code was (and is) overall not well arranged. That was why we looked firstly for some alternatives. There was a working web server (in uspace/apps/websrv) covering *socket*, *bind*, *listen*, *accept*, *recv*, *send* and *closesocket* functions for TCP sockets. It missed only the *connect* function. DNS IPC server dnsr (in uspace/srv/net/dnsr) used UDP sockets and covered *socket*, *sendto*, *recvfrom* and *closesocket* functions. Only the *bind* function for UDP sockets was missing.

We decided to put in order the nettest2 tool and execute it against a remote TCP ECHO server. It was necessary for *connect* function testing. Nettest2 used *sendto* and *recvfrom* calls on TCP sockets. It was naturally invalid because TCP communication is stateful. Except that, they did not initialize receive buffer length. The first *recvfrom* call thus always failed even on UDP. After nettest2 was working on both TCP and UDP, we made also the netecho tool working, but only under UDP protocol.

With those two fixes, we finally covered all functions of partial BSD socket interface. `Nettest2` supported TCP *socket*, *connect*, *recv*, *send* and *closesocket* functions. `Netecho` covered *socket*, *bind*, *recvfrom* and *sendto* functions on UDP sockets. We used the netcat utility to test `netecho` UDP server from another computer over Ethernet. Finally, we could run even `nettest2` against `netecho` on loopback (using UDP because `nettest2` works already on both transport protocols).

It was possible to check also fragmentation and reassembling in `inetsrv` with `netecho` UDP server. Reassembling is the biggest irregularity in IP datagrams processing. `Netecho` buffer has been enlarged from 1024 to 4096 bytes in order to overrun Ethernet MTU. Then we could send 3500-byte UDP messages with netcat to HelenOS and print its answers.

During that phase of preparation became known previous HelenOS implementation of IPv4 fragmentation and reassembling was never tested. Some bugs could appear later with the new implementation of linked-lists. Linked-list stores partial datagrams divided into fragments. Many bugs were, however, so serious that the implementation could not be working ever before. Among such bugs belongs unique assignment of identifiers to single fragments (not to whole datagrams). It caused that no two fragments could have the same identifier. Any operations adding received fragments to the partial datagram list was even missing. Received fragments always leaked and the list remained empty.

After IPv4 fragmentation and reassembling bug fixes, the testing set had completely covered outer interface of HelenOS networking stack. It included all major procedural variants of all single features. All their tests passed without errors. Previous implementation and its testing set were thus ready for IPv6 development.

4.3.3 Final testing set

Testing set thus contains:

1. `Ping` to loopback
2. `Ping` to a computer in the same network
3. `Ping` to a computer in a foreign network
4. Replies to ICMP *Echo request* messages from another computer
5. Creation, deletion and printing IPv4 networks and routes by `inet` tool
6. Setting IPv4 recursive DNS server by `dnscfg`
7. Translation of hostname to IPv4 address
8. `Websrv` on IPv4 correct functionality
9. `Nettest2` against a remote TCP ECHO server

10. Netcat from remote computer against `netecho` on HelenOS under UDP (Datagram size under MTU)

11. `Nettest2` against `netecho` under UDP on loopback

12. Netcat from remote computer against `netecho` on HelenOS under UDP (Datagram size over MTU)

and will be enlarged with:

13. `Ping6` to loopback

14. `Ping6` to a computer in the same network

15. `Ping6` to a computer in a foreign network

16. Replies to ICMPv6 *Echo request* messages from another computer

17. Creation, deletion and printing IPv6 networks and routes by `inet`

18. Setting IPv6 recursive DNS server by `dnscfg`

19. Translation of hostname to IPv4 by IPv6 recursive DNS server

20. Translation of hostname to IPv6 by IPv4 recursive DNS server

21. Translation of hostname to IPv6 by IPv6 recursive DNS server

22. `Websrv` on IPv6 correct functionality

23. `Nettest2` against a remote TCP over IPv6 ECHO server

24. `Nettest2` against a remote TCP over IPv4 ECHO server using *PF_INET6* sockets (mapping)

25. Netcat from a remote computer against `netecho` on HelenOS under UDP over IPv6

26. Netcat from a remote computer against `netecho` on HelenOS under UDP over IPv4 using *PF_INET6* listen socket (mapping)

27. `Nettest2` against `netecho` under UDP over IPv6 on loopback

28. `Nettest2` against `netecho` under UDP over IPv4 using *PF_INET6* sockets (All three variants of mapping - only `nettest2`, only `netecho` and both)

29. Netcat from a remote computer against `netecho` on HelenOS under UDP over IPv6 (Datagram size over MTU - IPv6 fragmentation)

4.4 Development phases

Development phases are steps transferring HelenOS networking stack from one consistent state to another one. The new state must be naturally nearer to the final implementation. Between two development phases, the state must be consistent. It means that actual testing set must be perfectly working.

4.4.1 Preliminary phases

First few phases were used just for accommodation of previous IPv4 implementation to be more easily transferrable to final IPv6-capable implementation. We changed all IPC transfers of IPv4 addresses to use IPC interface for passing larger data blocks. IPv4 addresses had only 4 bytes and were thus perfectly suitable as parameters of simple messages. Data blocks were, however, easily substitutable with 16-byte data blocks carrying already IPv6 addresses.

Secondly, we completely wiped out the *addr* library placed in `uspace/lib/c/generic/net/addr.c` and replaced it with correct implementation of standard *inet_pton* and *inet_ntop* functions. Only the `inet` tool needed to parse IP addresses with their netmasks. We implemented functions for parsing and printing IP addresses with netmasks there locally. All other occurrences of *addr_parse* and *addr_print* functions were replaced with matching *inet_ntop* and *inet_pton* calls.

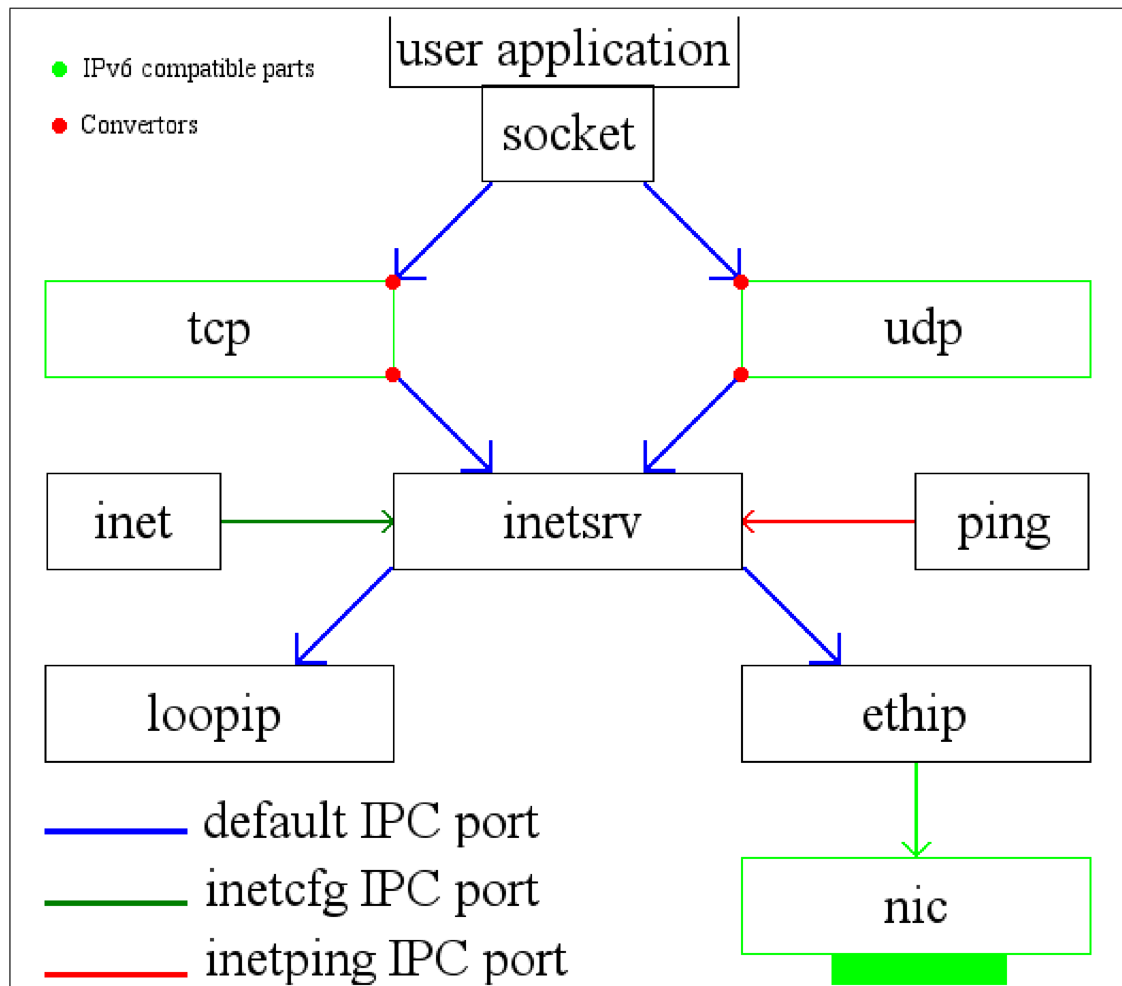
In the third phase we unified IPv4 address representation. Before that each task had used its own IPv4 address representation (socket library used *in_addr_t*, `tcp` used *netaddr_t*, `inetsrv` used *inet_addr_t*, `ethip` used *iplink_srv_addr_t*, etc.). We replaced all those data types with standard *in_addr_t* structure containing one *uint32_t* member *s_addr*. Simultaneously we defined macro value *INADDR_ANY* and replaced all other macros and numbers representing the same value.

In the fourth phase, we unified also the endianity of IPv4 addresses. IPv4 addresses had been previously stored in native endian format competing with intended IPv4 to IPv6 mapping. Since then, all IPv4 addresses were stored classically in big endian (network byte order) and no byte-order conversion of IP addresses was ever since needed.

4.4.2 Porting to IPv6

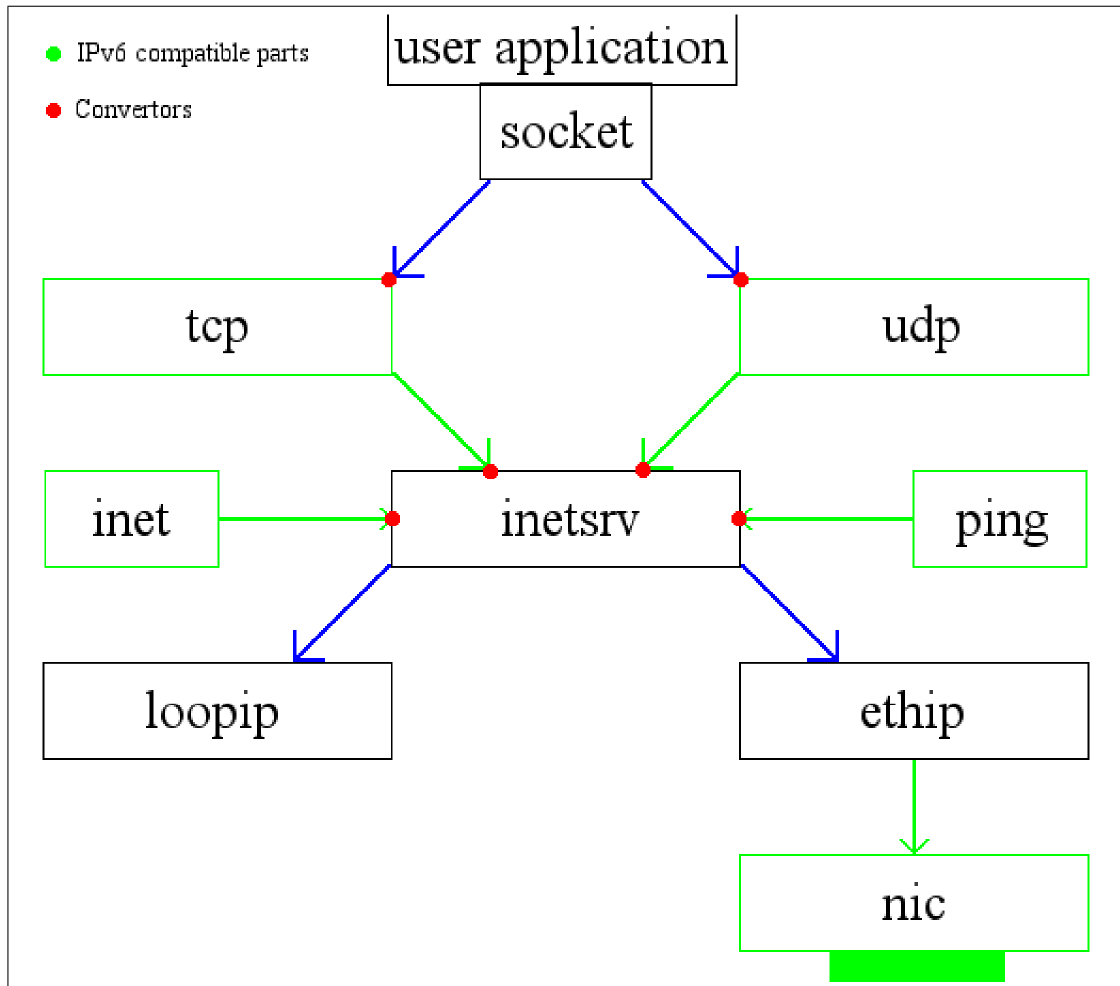
In the fifth phase we defined constants such as *in6addr_any*, macros and utilities described in the *Analysis* chapter. Then we ported `tcp` task to use and process IPv6 addresses. Both IPC interface handlers (with `inetsrv` and with socket library) used conversions back from and to IPv4. In the next phase, we did the same with the `udp` task.

20. HelenOS networking stack after the 6th Phase



Consequently, we modified the `inet` tool to be IPv6 capable. It automatically covered testing point 15 as described in the previous section. Then we changed interfaces of `inetsrv` IPC ports `inet`, `inetcfg` and `inetping` to use everywhere 16-byte IPv6 addresses. `Inetsrv` itself was even not IPv6 capable but it performed the conversions from IPv4-mapped IPv6 addresses to IPv4 by itself. `Ping` utility was accommodated to use new `inetping` interface with IPv6 addresses.

21. HelenOS networking stack after the 9th Phase



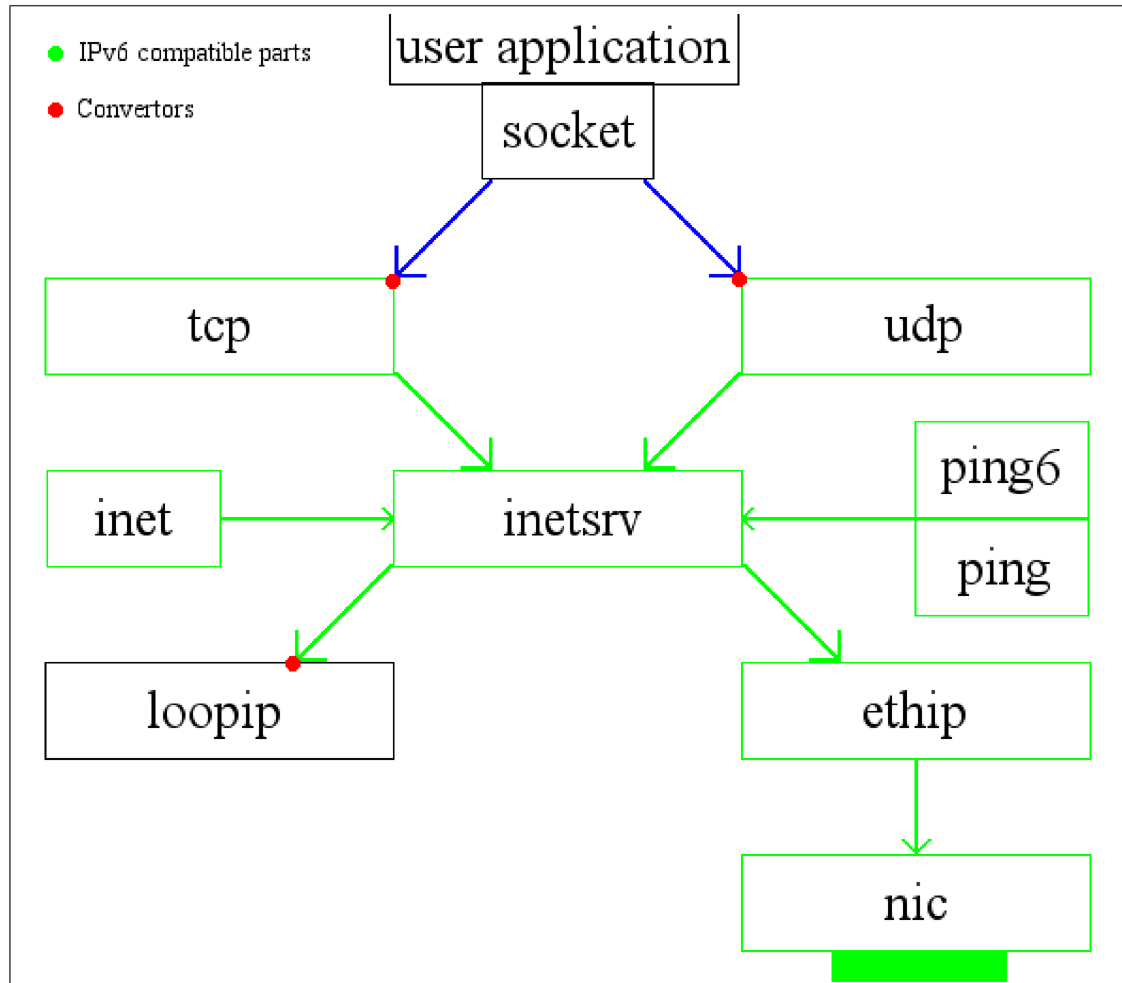
Next phase brought the largest changes. It was necessary to replace all IPv4 data structures in `inetsrv` task with their IPv6 alternatives. Later we added IPv6 PDU processing, distinction mechanisms, ICMPv6 processing and NDP protocol implementation. Immediately after that, we enhanced IPC interface between `inetsrv` and `ethip` as described in the *Analysis* chapter. Finally, we added IPv6 support also to `ethip`.

In that moment it was possible to let IPv6 traffic into the HelenOS networking stack and to start processing it. This processing began with receiving NDP solicitations and sending solicited NDP advertisements. First significant complication became known when the NIC driver did not accept any multicast-addressed Ethernet frames. We had to temporarily hack the `e1k` driver and in fact turn on multicast promiscuous mode to receive IPv6 traffic. *Neighbor solicitation* messages do not ever manage without multicast Ethernet addressing.

When solicited NDP advertisements were working correctly, we added `ping6` utility with the same behavior as `ping` just using IPv6 and ICMPv6 protocols.

It enlarged the testing set to cover all features of the final outer interface except points 13, 21 and 22.

22. HelenOS networking stack after the 13th Phase



When `ping6` worked actively and HelenOS correctly replied ICMPv6 *Echo request* messages, we made `loopip` compatible with IPv6 too. The next goal was socket library adaptation. Socket library was extended with new socket families, socket address families, conversions and compatibility checks. Socket library did not act correctly when sending data blocks larger than 4096 by *recv* or *send* functions. Error detection was almost missing and deadlock-prone. Nevertheless, socket library worked enough to test the new IPv6 implementation. Its rewrite was neither intended nor inevitable during this project.

We patched `websrv` application in order to test socket library IPv6 support using dual stack listening socket. `Netecho` and `nettest2` utilities supported *AF_INET6* sockets and socket addresses already since earlier.

When `websrv`, `netecho` and `nettest2` were working with both native IPv6 addresses and IPv4-mapped IPv6 addresses, we started to modify the DNS stack.

Three modifications were necessary:

1. Communication with IPv6 DNS servers
2. *AAAA* record support
3. Enhancement of lookup interface to differentiate IPv4 (*A* record) and IPv6 (*AAAA* record) lookups

Lastly we implemented the IPv6 fragmentation and reassembling capability. We introduced strong typing of socket families. Socket family assigned by *socket* function or inherited by *accept* function should be equal to socket address family used by *connect*, *bind* or *sendto* functions.

5. Evaluation

Neither special testing nor debugging phase was needed because of continual testing thanks to salami-tactics use. We did only some measurements verifying that the new implementation does not drag behind the original IPv4 implementation from the performance point of view.

5.1 Functional aspects

New networking stack recognizes Ethernet frames carrying IPv6 datagrams. It can assign IPv6 addresses with their netmasks to single interfaces and list or delete them. Also the assignment of IPv6 routing table lines is possible but the router IP must be in the same network as the source IP of outgoing datagrams. IPv6 implementation supports correct receiving of IPv6 datagrams destined to one of local unicast addresses or to solicited-node multicast addresses. It correctly sends and routes outgoing datagrams. *Echo request*, *Echo reply*, *Neighbor solicitation* and *Neighbor advertisement* ICMPv6 messages are correctly processed. It includes automatic replies, pinging and NDP translations.

TCP over IPv6, UDP over IPv6 and socket interface providing *AF_INET6* sockets and socket addresses are also supported. *SOCK_DGRAM* sockets provide *socket*, *bind*, *sendto*, *recvfrom* and *closesocket* functions. *SOCK_STREAM* sockets provide *socket*, *bind*, *listen*, *accept*, *connect*, *send*, *recv* and *closesocket* functions. Their features include all features of *AF_INET* sockets. `Inetsrv` task supports even full IPv6 fragmentation and reassembling. DNS resolution is able to use IPv6 DNS servers and to resolve *AAAA* records (hostname to IPv6 address translation). Loopback interface also supports IPv6 communication on the same level as Ethernet interfaces.

IPv6 implementation supports dual stack sockets with IPv4-mapped IPv6 addresses. Link-local addresses are automatically assigned to each NIC during start-up.

For day-to-day use, HelenOS IPv6 stack still lacks IPv6 address scoping and related features. All IPv6 addresses (together with all IPv4 addresses) belong to one global scope. It means they are valid and supposed to be unique everywhere. Link-local addresses are not differentiated and their destinations are thus ambiguous when more NICs are present. Finally, routers are stored using their IPv6 addresses and not their MAC addresses and NIC interfaces which is usual.

To be working correctly and usually, OS networking stack must support at least three IPv6 address scopes: link-local, site-local and global scope. Link-local IPv6 addresses (derived from MAC address and assigned automatically) should also enforce concrete network interface to be specific enough. All link-local addresses belong to the same network. If no concrete interface of a link-local address is defined, OS has in fact no idea what interface to use. `Ping6` utility should support

% operator separating destination IPv6 address and NIC identifier. Standard OS users can set a link-local address as their default router. HelenOS is missing that feature by now. If a link-local address represents a router, HelenOS automatically uses its own link-local address as the source IP in outgoing datagrams. Global IPv6 destinations are naturally unable to route such traffic back. HelenOS users must use a global IPv6 address as their default router if they want to communicate over Internet.

5.2 Performance aspects

As prototype implementation, HelenOS IPv6 stack uses almost no optimizations. Its performance is therefore relatively low in comparison with IPv6 stacks of other operating systems. When compared with HelenOS IPv4 stack, HelenOS IPv6 stack achieves practically the same performance. It implies that the new code probably does not contain any substantial performance defects.

23. Performance comparison

HelenOS IPv4 stack	**HelenOS IPv6 stack**
Qemu-x86_64, E1000 TAP on Linux	Qemu-x86_64, E1000 TAP on Linux
Webserver - flow test http://10.0.2.15:8080/test Downloading 300kB file with wget 1. **11,9K**=25s 2. **12,9K**=23s 3. **12,0K**=25s 4. **11,6K**=26s 5. **11,2K**=27s 6. **11,6K**=26s 7. **12,5K**=24s 8. **10,4K**=29s 9. **12,1K**=25s 10. **13,8K**=22s 11. **13,1K**=23s 12. **13,4K**=22s 13. **12,3K**=24s 14. **11,3K**=27s 15. **12,5K**=24s 16. **10,7K**=28s Min: 10,4kB/s Avg: **12,0kB/s** Max: 13,8kB/s	Webserver - flow test http://[fc02::2]:8080/test Downloading 300kB file with wget 1. **12,2K**=25s 2. **10,4K**=29s 3. **11,6K**=26s 4. **12,8K**=23s 5. **11,4K**=26s 6. **12,6K**=24s 7. **12,9K**=23s 8. **10,4K**=29s 9. **12,0K**=25s 10. **13,6K**=22s 11. **13,8K**=22s 12. **10,4K**=29s 13. **10,3K**=29s 14. **11,5K**=26s 15. **10,4K**=29s 16. **12,7K**=24s Min: 10,3kB/s Avg: **11,8kB/s** Max: 13,8kB/s
Ping - latency test PING 10.0.2.15 56(84) bytes of data. — 10.0.2.15 ping statistics — 256 packets transmitted, 256 received, 0% packet loss, time 255277ms rtt min/avg/max/mdev = 16.485/**26.702**/40.263/4.575 ms	Ping - latency test PING fc02::2 56 data bytes — fc02::2 ping statistics — 256 packets transmitted, 256 received, 0% packet loss, time 255350ms rtt min/avg/max/mdev = 18.319/**26.403**/42.662/4.811 ms

6. Future & related work

6.1 Future work

How to proceed with the development was described in the previous chapters. Among immediate goals belong refactoring the socket library, adding regular support of Ethernet multicasting to all NIC drivers (not only to E1000), IPv6 address scoping and routing by MAC addresses. There are even many others IPv6 features that are not very complicated to implement. HelenOS IPv6 stack is well designed, logically structured and highly distributed. Its future extensions are to be smooth and awaited.

6.1.1 Scopes support and routing mechanism upgrade

Scopes implementation should be trivial. The *in6_naddr_t* structure will be extended with another integer expressing IP address scope. If not set alternatively, `fe80::/10` addresses belong to link-local scope, `fc00::/7` addresses belong to site-local scope, `::1/128` to host-local scope and any other address belongs to global scope. *Inet_get_srcaddr* function will nevermore derive source IP address from the first-hop router IP. It just determines concrete NIC in the network of the first-hop router and chooses IP address belonging to the same scope as the destination.

Inet_sroute_info_t structure will look as follows:

24. New routing table row format

```
/** Static route info */
typedef struct {
    /** Destination network address */
    in6_naddr_t dest;
    /** Static route name */
    char *name;
    /** Router family */
    uint16_t family;

    /** Router definition */
    union {
        /** Router IPv4-mapped IPv6 address */
        in6_addr_t v4router;
        struct {
            /** Router MAC address */
            uint64_t hwaddr;
            /** Router link name */
            char * link_name;
        } v6;
    }
} inet_sroute_info_t;
```

IPv4 routers will be still defined by their IP addresses. IPv6 routers will be defined by their MAC addresses and NIC interfaces.

6.1.2 Automatic IPv6 address and default router assignment

Next concrete goal, following scopes support and routing table upgrade, is the automatic configuration of routers. NDP module should receive even *Router advertisement* messages (ICMPv6 type `135`) and eventually send *Router solicitation* messages (ICMPv6 type `134`). *Router advertisement* messages should be processed in order to assign global IPv6 address to the interface that received the message. Newly assigned IPv6 address should be derived from received routing prefix (first 64 bits) and from the tail of the link-local address (last 64 bits). Router MAC address should be taken directly from the advertisement message. It is not necessary to resolve it by NDP again.

The networking stack could be in two states (with the state stored as a global synchronized boolean variable). Either it is configured or it is not. If it already

has set an address and a router, it does not send any *Router solicitation* messages and ignores all *Router advertisement* messages. Otherwise, it periodically sends *Router solicitation* messages and is able to process a *Router advertisement* message with an immediate change of its state.

6.1.3 Socket library

Networking modes, ipv4-only and ipv6-only, should be propagated up to the socket library. Socket library should enforce already the *socket* function to fail when called with prohibited socket family. Similarly it should immediately detect and forbid IPv4-mapped IPv6 addresses in *sockaddr_in6* structure (in *bind*, *connect* and *sendto* functions) if the IPv4 traffic is prohibited. That propagation will lead to better error detection.

Next good feature of *PF_INET6* sockets is the *IPV6_V6ONLY* option set by *setsockopt* and read by *getsockopt* functions. After the socket library refactoring, such option can be easily introduced modifying just `tcp_socket_match` function in `tcp` task and `udp_socket_match` function in `udp` task.

6.1.4 ICMPv6 error messages and MTU discovery

ICMPv6 error messages are generally more important than ICMP error messages. It is caused by fragmentation restrictions. Each router that cannot forward an IPv6 datagram sends appropriate *Packet too big* message back to the sender and discards that datagram. The sender must process such message and perform Path MTU discovery (PMTUD) to shrink all next datagrams sent to the same destination.

Path MTU discovery in IPv6 networks is much more important than in IPv4 networks. IPv6 packet sender must strictly follow maximal MTU of sent datagrams. No routers along the way can fragment its datagrams. Datagrams bigger than maximal path MTU must be necessarily discarded by the specification of the IPv6 protocol.

ICMPv6 module should therefore recognize and accept also *Packet too big* messages and hold a list with destination IPv6 addresses and their MTU values. This list should contain all IPv6 destination addresses if their path MTU is lower than 1500 bytes. IPv6 PDU encoding must obey the new MTU and create fragments of correct sizes.

6.2 Related work

This section compares HelenOS with two other well-known microkernel operating systems from the IPv6 capability point of view.

6.2.1 Hurd IPv6 implementation

GNU Hurd is a microkernel operating system. It is still in development and it is even older than HelenOS (since 1998). GNU Hurd has completely different approach of its networking stack. Its networking stack is concentrated in one process (system service) called `pfinet`. `Pfinet` contains completely adopted networking stack with its IPC adapter. The networking stack is adopted from the Linux kernel, concretely from deprecated 2.4.x releases.

Comparison of Hurd and HelenOS IPv6 implementations is therefore almost impossible because of completely different approach. Hurd IPv6 stack is adopted from Linux kernel. Linux is a monolithic OS naturally not honoring separation of system servers. It integrates whole code together to be executed originally even in kernel processor mode. It is able to enuntiate that HelenOS implementation of IPv6 is much more faithful to microkernel methodology of OS development. Hurd has contrarily more IPv6 features because of Linux-kernel code adoption.

6.2.2 Minix IPv6 implementation

Minix is also a microkernel OS. It provides multiple variants of networking. The main variant is the `inet` service integrating (similarly to Hurd) whole networking stack into one process (including all Ethernet, loopback, TCP and UDP). `Inet` service does not support IPv6 at all.

Alternative technology to native `Inet` service is the `lwIP` stack. Actually, `lwIP` can use either IPv4 or IPv6 but not both. Dual stack operation mode is not yet supported[1]. In these days, all Internet users inevitably need IPv4 connectivity. Majority of servers providing Internet content are not yet IPv6 connectable. It means that IPv6 connectivity without IPv4 connectivity is unavailing. It is good at best for testing purposes. HelenOS is thus much more IPv6 capable than Minix.

[1]IPv6 lwIP Wiki, on-line at `http://lwip.wikia.com/wiki/IPv6`

7. Conclusion

HelenOS finally supports IPv6 protocol and related technologies on the same level as IPv4 protocol. The two protocols have various differences described in previous chapters. Features of both implementations are analogous. New implementation respects microkernel multiserver architecture of HelenOS. It allows three networking modes - dual stack (which is the default), IPv6 only and IPv4 only.

All goals set for this thesis have been achieved. Some minor IPv6-related defects surviving in the networking stack from earlier became known. NIC drivers do not accept multicast frames. The socket library error-detection is buggy and deadlock-prone. Socket library is to be refactored as a whole later. Implementation of IPv6 and related technologies is of course multiplatform. Networking stack in HelenOS is a relatively high-level part of the operating system. The only portability problem was the endianity of network ports and of other multi-byte items in packet-, datagram-, fragment- or frame-headers.

IPv6 implementation still lacks address-scopes support. The routing mechanism does not obey address-scopes, which is a non-fatal complication. Link-local addresses are not differentiated by network interfaces. It causes incomplete support of multiple NICs connected to one computer. All these insufficiencies can be relatively smoothly resolved and the IPv6 implementation as a whole is well prepared to be extended by new features in the future.

Bibliography

[1] Berkeley sockets, on-line at https://en.wikipedia.org/wiki/Berkeley_sockets

[2] Broadcast, on-line at https://en.wikipedia.org/wiki/Broadcast_address

[3] HelenOS documentation, on-line at http://www.helenos.org/documentation

[4] HelenOS NIC framework documentation, on-line at http://www.helenos.org/doc/helnet.pdf

[5] HelenOS, on-line at http://www.helenos.org/

[6] IPC for dummies, on-line at http://trac.helenos.org/wiki/IPC

[7] IPv6 lwIP Wiki, on-line at http://lwip.wikia.com/wiki/IPv6

[8] Localhost, on-line at http://en.wikipedia.org/wiki/Localhost

[9] MAC address, on-line at http://en.wikipedia.org/wiki/MAC_address

[10] Microkernel, wiki, on-line at http://en.wikipedia.org/wiki/Microkernel

[11] Multicast, on-line at https://en.wikipedia.org/wiki/Multicast_address

[12] Network interface controller, on-line at http://en.wikipedia.org/wiki/Network_interface_controller

[13] Requirements for Internet Hosts - Communication Layers, RFC 1122, on-line at http://tools.ietf.org/html/rfc1122

[14] TCP/IP tutorial, RFC 1180, on-line at http://tools.ietf.org/html/rfc1180

[15] Path MTU discovery, RFC 1191, on-line at http://www.ietf.org/rfc/rfc1191.txt

[16] Definition of a socket, RFC 147, on-line at http://tools.ietf.org/html/rfc147

[17] The IP Network Address Translator (NAT), RFC 1631, on-line at http://www.ietf.org/rfc/rfc1631.txt

[18] Variable Length Subnet Table For IPv4, RFC 1878, on-line at http://tools.ietf.org/html/rfc1878

[19] Address Allocation for Private Internets, RFC 1918, on-line at http://tools.ietf.org/html/rfc1918

[20] Guidelines for creation, selection, and registration an Autonomous System (AS), on-line at http://tools.ietf.org/html/rfc1930

[21] Internet Protocol version 6, specification, RFC 2460, on-line at http://tools.ietf.org/html/rfc2460

[22] Transmission of IPv6 Packets over Ethernet Networks, RFC 2464, on-line at http://tools.ietf.org/html/rfc2464

[23] Basic Socket Interface Extensions for IPv6, RFC 2553, on-line at http://www.ietf.org/rfc/rfc2553.txt

[24] Application Aspects of IPv6 transition, RFC 4038, on-line at http://tools.ietf.org/html/rfc4038

[25] IP Version 6 Addressing Architecture, RFC 4291, on-line at http://tools.ietf.org/html/rfc4291

[26] Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, on-line at http://tools.ietf.org/html/rfc4443

[27] Neighbor Discovery for IP version 6 (IPv6), RFC 4861, on-line at http://tools.ietf.org/html/rfc4861

[28] IPv6 Stateless Address Autoconfiguration, RFC 4862, on-line at http://tools.ietf.org/html/rfc4862

[29] IPv6 Node Requirements, RFC 6434, on-line at http://tools.ietf.org/html/rfc6434

[30] User Datagram Protocol, RFC 798, on-line at http://www.ietf.org/rfc/rfc768.txt

[31] Internet Protocol, RFC 791, on-line at http://www.ietf.org/rfc/rfc791.txt

[32] Internet Control Message Protocol, RFC 792, on-line at http://tools.ietf.org/html/rfc792

[33] Transmission Control Protocol, RFC 793, on-line at http://www.ietf.org/rfc/rfc793.txt

[34] An Ethernet Address Resolution Protocol, RFC 826, on-line at http://tools.ietf.org/html/rfc826

[35] A Standard for the Transmission of IP Datagrams over Ethernet Networks, RFC 894, on-line at http://tools.ietf.org/html/rfc894

List of Abbreviations

- A-class network - network with 8-bit netmask containing thus 2^{24} IPv4 addresses
- ACK - TCP acknowledgment flag
- ARP - Address Resolution Protocol
- BGP - Border Gateway Protocol
- BSD - Berkeley Software Distribution
- CPU - Central Processing Unit
- DHCP - Dynamic Host Configuration Protocol
- EINVAL - Invalid value Error
- I/O - Input/Output
- ICMP - Internet Control Message Protocol (for IPv4)
- ICMPv6 - Internet Control Message Protocol version 6 (for IPv6)
- IGMP - Internet Group Management Protocol
- IP - Internet Protocol
- IPv4 - Internet Protocol version 4
- IPv6 - Internet Protocol version 6
- IPC - Inter-process communication
- MAC - Media Access Control
- MTU - Maximum transmission unit
- NAT - Network Address Translation
- NDP - Neighbor Discovery Protocol
- NIC - Network interface controller
- OS - Operating system
- OSPF - Open Shortest Path First
- PDU - Protocol data unit
- PMTUD - Path MTU Discovery
- POSIX - Portable Operating System Interface
- QoS - Quality of Service

- RST - TCP reset flag
- SYN - TCP synchronization flag
- TCP - Transmission Control Protocol
- TTL - Time to live
- UDP - User Datagram Protocol