

ip46nat User's Guide

Author: Tomasz Mrugalski
version 2008-10-05

Table of contents

1	Project overview.....	4
1.1	Phase 1: IPv4 to IPv6 NAT.....	4
1.2	Phase 2: IPv4 over IPv6 tunneling.....	4
2	Project status.....	4
2.1	Latest status.....	4
2.2	Revision history.....	4
3	Installation.....	5
3.1	Connecting LinkSys.....	5
3.2	Firmware upgrade (using original web interface).....	6
3.3	Firmware upgrade (using linux console).....	7
3.4	Firmware upgrade (TFTP).....	8
3.5	Enabling boot_wait phase.....	9
3.6	First connection.....	9
3.7	ipk packages.....	10
3.8	Package installation.....	10
4	Network configuration.....	11
5	Manual (static) configuration.....	12
5.1	ip46nat: IPv4-to-IPv6 NAT Overview.....	12
5.2	ip46nat: IPv4 to IPv6 NAT operation.....	13
5.3	ip46nat: IPv6 to IPv4 NAT operation.....	13
5.4	Firewall – Limiting extra traffic.....	13
5.5	IPv4 over IPv6 tunneling.....	14
5.6	DNS proxy.....	15
5.7	Router Advertisements daemon.....	16
6	Remote configuration via DHCPv6.....	16
6.1	Dibbler installation.....	16
6.2	Using dibbler to control NAT (obsolete).....	18
6.3	Remote traffic type control via DHCPv6 (Tunnel-mode).....	18
6.4	Tunnel-Mode operation.....	19
6.5	notify script.....	20
6.6	Possible enhancements.....	20
6.7	Client configuration.....	21
6.8	Server configuration.....	21
6.9	Client-specific server configuration.....	22
7	Compilation.....	23
7.1	Linux for embedded devices (OpenWRT).....	23
7.2	ip46nat kernel module.....	24
7.3	ip46nat kernel module as an ipk package.....	25
7.4	dibbler software.....	25
7.5	dibbler software as an ipk package.....	26
7.6	IPv4-over-IPv6 tunnel modules.....	26
8	IPv4-to-IPv6 NAT (Phase 1) testing.....	27
8.1	IPv4 to IPv6 traffic.....	27
8.2	IPv6 to IPv4 traffic.....	29
8.3	Firewall configuration.....	31
8.4	Negative testing.....	31
8.5	Performance testing.....	32

8.6	Statistics.....	32
8.7	Test conclusion.....	32
9	IPv4 over IPv6 tunneling (Phase 2) testing.....	32
9.1	Traffic validation.....	33
9.2	Remote ipip6 configuration validation.....	35
9.3	Remote ip46nat configuration validation.....	37
10	Source code.....	39
10.1	Code overview for ip46nat module.....	39
11	Best practices and debugging tips.....	39
12	Links.....	39
13	Contact.....	40

1 Project overview

Goal of this project is to provide solution for seamlessly forward IPv4 traffic over IPv6 networks. There are two possible migration modes: IPv4 to IPv6 NAT (see section 1.1 for details) and IPv4 over PPP over L2TP over IPv6.

This document describes usage, installation and configuration of the software required to setup and run remotely configurable router capable of handling two different post-IPv4 traffic types: IPv4 to IPv6 translation and IPv4 over IPv6 tunneling. In particular, new features of the Dibbler software (implementation of the DHCPv6 server and client) used for remote automatic configuration is described.

In particular, following software will be developed: enhancements to the Dibbler software, new kernel modules for IPv4 to IPv6 encapsulation and IPv4 over IPv6 tunneling, porting Dibbler code to LinkSys™ embedded device. Although developed software is delivered mainly in an easier to use compiled form, source code is also provided. This document describes procedures required to build toolchain used for cross-compilation, and the compilation of the developed code as well.

1.1 Phase 1: IPv4 to IPv6 NAT

First approach to the IPv4 over IPv6 network assumes that incoming IPv4 packets will be converted and forwarded as IPv6. IPv4 addresses will be "expanded" into full IPv6 addresses, using 2 extra prefixes: M and P. Returning IPv6 packets will be converted back to IPv4.

1.2 Phase 2: IPv4 over IPv6 tunneling

Second approach assumes that IPv4 packets will be tunneled over IPv6. Every IPv4 packet will be encapsulated in extra IPv6 header. Extra steps to configure IPv4-in-IPv6 tunnel must be provided, like routing configuration.

2 Project status

2.1 Latest status

For latest status, list of tasks already completed, work in progress and upcoming tasks, see project web page (<http://klub.com.pl/ip46nat/>). Should priorities change during code development, please contact Tomasz Mrugalski. As of 2008-09-02, both phases are fully functional.

Some of the features that were available during Phase 1, have been updated during Phase 2. In particular, kernel 2.6.23 was replaced with kernel 2.6.25.16. Also mappingprefix is considered obsolete and was superseded by notify script.

2.2 Revision history

Date	Version	Primary Author	Change description
2008-06-04	-	Tomasz Mrugalski	Initial documentation release
2008-06-22	-	Tomasz Mrugalski	Current status section removed (link to website provided instead), project overview updated, compilation process described (sections 6.1-6.5 added)

2008-07-02		Tomasz Mrugalski	Dibbler-NAT integration, phase 1 completion testing, extra description regarding device model.
2008-09-04		Tomasz Mrugalski	IPv4-over-IPv6 tunneling, dibbler-tunnel mode, phase 2 testing
2008-09-14		Tomasz Mrugalski	Traffic types explained, source code walkthrough added.
2008-10-05		Tomasz Mrugalski	DNS traffic handling section added

3 Installation

To complete installation, several steps are required. Following sections describe, how to achieve specific steps. In general, following actions are required:

1. Install Linux on LinkSys WRT54 device (or similar)
2. Install ip46nat module (required for IPv4-to-IPv6 NAT)
3. Install ip6_tunnel (required for IPv4-over-IPv6 tunneling)
4. Install and setup dibbler-client for remote configuration.
5. Install dibbler-server on a PC. Server will provide configuration details for the client.

It is also possible to install required software (i.e. modified Linux kernel + modules + dibbler) on a PC machine and use it to perform IPv4-IPv6 NAT or IPv4-over-IPv6 tunneling. This optional scenario may be used as an validation or debugging environment.

3.1 Connecting LinkSys

Before any configuration or firmware modification, make sure that you have full connectivity to your LinkSys device. LinkSys devices by default use 192.168.1.1 address, so to communicate with it, another address from 192.168.1.0/24 pool is required. For example, PC may be configured to use 192.168.1.100/24 address. To check if you have connectivity with LinkSys device, use following command:

```
ping 192.168.1.1
```

Make sure that you have LinkSys connected using the rightmost socket. See Fig.1 below.



Fig. 1: Connecting LinkSys to LAN

Note: IPv4 address set to 192.168.1.1 is a LinkSys' default setup. It reverts to this configuration after every firmware upgrade. Also it is its default factory configuration.

3.2 Firmware upgrade (using original web interface)

Before attempting to install Linux on LinkSys device, make sure that this particular model is supported. Please consult <http://wiki.openwrt.org/TableOfHardware> . Note that even small deviations are important. Sometimes version 1.0 and 1.1 are quite different. See Fig. 2 for example how to check your particular model. In general, at least 4MB flash and 16MB ram is required.



Fig. 2: LinkSys device model check

Linux installation on Linksys is being performed as a firmware upgrade. During the first installation, original web interface (provided by LinkSys) should be used. From PC using the same address space (see section 3.1), use web browser to connect to your LinkSys web interface. See Fig.3 below.

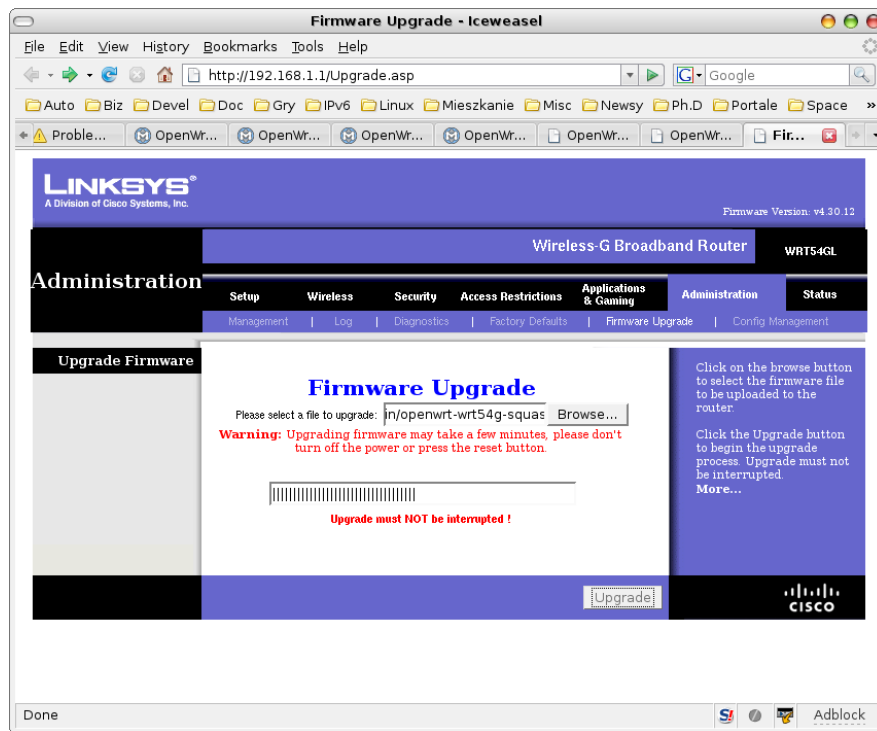


Fig. 3: Firmware upgrade using original web interface

Select appropriate firmware image (i.e. file that ends with `-squashfs.bin` and is corresponding to the name of used device). There are some sanity checks in the firmware upgrade procedure, but using wrong image may result in rendering the router unusable. You may want to check supported hardware list: <http://wiki.openwrt.org/TableOfHardware?action=show&redirect=toh> Please verify that you are using proper firmware. After firmware was uploaded, flashing takes place. It can take up to 2 minutes. After completion, router will reboot. Make sure to not reboot or power off the router before it finishes flashing.

It also may be beneficial to read following installation guide: <http://wiki.openwrt.org/InstallingWrt54gl>

3.3 Firmware upgrade (using linux console)

Firmware upgrade from OpenWRT (i.e. when your LinkSys device was flashed already) is done by using `mtd` tool. Copy `openwrt-brcm47xx-squashfs.trx` file to the `/tmp` directory. Note that this is a different image file than used in the web interface. Copy it to your LinkSys device:

```
scp openwrt-brcm47xx-squashfs.trx root@192.168.1.1:/tmp
```

This command will copy required firmware image to `/tmp` directory. Change to that directory and begin flashing, using following command:

```
cd /tmp
mtd -r write openwrt-brcm47xx-squashfs.trx linux
```

After flashing is complete, device will reboot. It takes up to 2 minutes to finish flashing and

rebooting. Please note that after such firmware upgrade, all possible changes made to the router configuration will be lost. That includes all software packages installed and all configuration changes.

Note: .bin and .trx firmware image files contain the same image, but .trx is a "raw" image, while .bin has extra headers for the purpose of being recognized as a valid image by the original web interface.

Note: mtd is a command-line tool. As most of the OpenWRT software it may come pre-installed or as a separate package. If the mtd is missing, simply install mtd_7_mipsel.ipk package.

3.4 Firmware upgrade (TFTP)

Another way to install new firmware is to use TFTP protocol. When boot_wait phase is enabled (see section 3.5), it is possible to upload new firmware using TFTP protocol. On a Linux box that uses IPv4 address from the same class (see section 3.1), use TFTP client to send firmware. For example:

```
tftp 192.168.1.1
tftp>binary
tftp>rexmt 1
tftp>timeout 60
tftp>trace
Packet tracing on.
tftp> put openwopenwrt-wrt54g-2.4-squashfs.bin
```

In the example above, client will try to send firmware at 1 second intervals. It will give up after 60 seconds. After those commands are typed, shut down and then boot your device. Following messages should be displayed:

```
$tftp 192.168.1.1
tftp> binary
tftp> rexmt 1
tftp> timeout 120
tftp> trace
Packet tracing on.
tftp> put openwrt-wrt54g-squashfs.bin
sent WRQ <file=openwrt-wrt54g-squashfs.bin, mode=octet>
sent WRQ <file=openwrt-wrt54g-squashfs.bin, mode=octet>
sent WRQ <file=openwrt-wrt54g-squashfs.bin, mode=octet>
sent WRQ <file=openwrt-wrt54g-squashfs.bin, mode=octet>
sent WRQ <file=openwrt-wrt54g-squashfs.bin, mode=octet>
sent WRQ <file=openwrt-wrt54g-squashfs.bin, mode=octet>
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ACK <block=1>
sent DATA <block=2, 512 bytes>
...
received ACK <block=3592>
sent DATA <block=3593, 32 bytes>
received ACK <block=3593>
Sent 1839136 bytes in 15.1 seconds
```

Note: bin images are expected to be uploaded via TFTP, not the raw trx images. Attempt to upload trx image (or in fact any other file that does not have proper headers) will cause

bootloader to reject the file:

```
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ERROR <code=4, msg=code pattern incorrect>
Error code 4: code pattern incorrect
```

3.5 Enabling boot_wait phase

Most embedded devices have multi-stage boot process. After the device is powered up, it runs bootloader. Its main task is to check if flash memory contains proper firmware, load and run it. Broken, corrupted or non-functional firmware may cause the endless loop of firmware loading and reboots or device crash. This state of device being unusable is often referred to as “bricked”. There are basically 2 methods of recovering such device:

1. Use JTAG connector to upload new firmware
2. Use TFTP to upload new firmware via network

As the first option requires hardware modification (most if not all LinkSys devices come without JTAG connector, so soldering and extra JTAG cable is necessary), it is much easier to use TFTP transfer.

Bootloader may be configured to wait specified period for incoming TFTP packets. However, this waiting phase delays device boot, so vendors often disable this feature. Fortunately, it can be reenabled. To enable boot_wait phase, use following commands from the command-line:

```
nvrn set boot_wait=on
nvrn commit
```

Note: this feature works under Linux kernels 2.4 only. For that purpose, you may want to use any stable OpenWRT firmware that is kernel-2.4 based. Once boot_wait is enabled, it is safe to experiment with new firmware images (kernel 2.6 based for example). One way to obtain 2.4 based firmware is from OpenWRT homepage: <http://downloads.openwrt.org/kamikaze/7.09/>

3.6 First connection

After performing firmware upgrade, it is possible to connect to the router using telnet command. Please run following command: telnet 192.168.1.1 from a PC console. See fig. 3 for example session. There is no root password. Please change root password by issuing passwd command. After this operation, telnet service will be disabled. SSH will be enabled instead. Note that ssh requires some time (~30 seconds) to generate keys, so it will reject any connection attempts at that time.

- kernel_2.6.25.16 – dummy package that is required by other packages.
- mtd - mtd is a tool used to flash router. That is the preferred way to flash router, once Linux have been installed.
- ip - powerful tool used for network configuration. Part of the iproute2 suite. For example, interfaces, addresses and tunnels are configured using this tool.
- iptables - optional package, may be used to configure IPv4 firewall and/or IPv4 only NAT.
- ip6tables - optional package, may be used to configure IPv6 firewall and/or IPv6 only NAT.
- uclibcxx - C++ library required to run all software written in C++, e.g. dibbler software
- dibbler-client - DHCPv6 client, used to retrieve configuration and configure the device.
- dibbler-server - DHCPv6 server, used to distribute addresses and configuration parameters to other nodes.

Kernel modules are also handled as ipk packages. To distinguish between user-space software and kernel modules, the latter use kmod- prefix. Here is a list of useful kernel modules:

- kmod-ipv6 – module that provides IPv6 capability.
- kmod-ip46nat – provides IPv4-to-IPv6 NAT functionality.
- kmod-ip6-tunnel – provides IPv4 over IPv6 tunneling. Requires iptunnel6 module to be present.
- kmod-iptunnel6 – this module is required by ip6-tunnel. It also requires IPv6 module.
- iptables – firewall and advanced routing tool
- ip6tables – IPv6 version of the iptables

For information how to compile additional software, see section 7.

Also keep in mind available space on the device. Use df -h command, if necessary. According to OpenWRT homepage, when base partition is full, various errors start to appear and firmware may get corrupted.

4 Network configuration

Before attempting to perform configuration, it is strongly recommended to be familiar with the following guide: <http://wiki.openwrt.org/OpenWrtDocs/NetworkInterfaces>

All LinkSys WRT routers use one common Ethernet interface, duplicated using different vlans. In the latest OpenWRT version, that is being reported as eth0, eth0.0, eth0.1. Ports 1-4 are grouped together and named br-lan. Note that this name have been changed during OpenWRT development, so older tutorial available on the Internat may use different name. To see or modify interface names, see `/etc/config/network`.

To enable IPv4 and IPv6 forwarding, use following command:

```
echo 1 > /proc/sys/net/ipv4/conf/all/forwarding
echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

To configure routing, ip command may be used. Assuming that we want to NAT traffic from 192.168.1.0/24 to 2000::/64, following command may be used:

```
ip route add 192.168.1.0/24 dev eth0.0
ip route add 2000::/64 dev eth0.1
```

In case of WRT54GL v1.1 (the model used by author), rightmost socket (next to power supply) is called eth0.0 (that is the interface that has default 192.168.1.1 address assigned). Another interface, labeled as WAN, is called eth0.1.

5 Manual (static) configuration

This section describes how to achieve several aspects of the LinkSys configuration in a manual way. Manual means a static, local configuration performed by locally issued commands or scripts. For automatic, remote, DHCPv6-based configuration, see section 6.

5.1 *ip46nat*: IPv4-to-IPv6 NAT Overview

To perform IPv4-to-IPv6 NAT, a separate kernel module have been developed. Although it would be possible to achieve similar functionality in the user space, kernel module provides the best efficiency. For the easiness of installation, it is being distributed as a ipk package. Please install it as any other ipk packages:

```
ipkg install kmod-ip46nat_2.6.25.16-brcm47xx-1_mipsel.ipk
```

After installation is complete, ip46nat kernel module may be loaded, using following command:

```
insmod /lib/modules/2.6.25.16/ip46nat.ko v6prefixm=2000:: v6prefixp=3000::
v4addr=10.10.1.0
```

Module reports its operation using normal kernel messages. To see kernel output, dmesg command may be used. It also appears to be useful to filter dmesg output using tail command. For debugging purposes, ip46nat module prints information about every incoming packet and configured pattern.

After module is loaded, it will start printing information about all received IPv4 and IPv6 traffic. Packets that match configured criteria are marked using *. When match is found, packet will be recoded and transmitted.

To see last 10 messages, use following command:

```
dmesg | tail -10
```

After module is loaded, it reports operation readiness and begins to filter incoming traffic immediately:

```
IPv4-IPv6 NAT module loaded: v6prefixp: 3000::, v6prefixm: 2000::, v4addr: 10.10.1.0
Handlers for IPv4 and IPv6 installed.
#IPv4 rcvd (rcvd so far: 1) [src=192.168.1.100, dst=192.168.1.1,looking for 10.10.1.0/24]
#IPv4 rcvd (rcvd so far: 2) [src=192.168.1.100, dst=192.168.1.1,looking for 10.10.1.0/24]
#IPv4 rcvd (rcvd so far: 3) [src=192.168.1.100, dst=192.168.1.1,looking for 10.10.1.0/24]
```

Kernel module may be unloaded at any time. To do so, use command:

```
rmmod ip46nat
```

After kernel is unloaded, statistics are being presented. To see them, use dmesg command.

```

Handlers for IPv4 and IPv6 removed.
---IPv4-IPv6 NAT statistics-----
IPv4-to-IPv6 packets: 0
IPv6-to-IPv4 packets: 0
IPv4 rcvd: 74, sent: 0
IPv6 rcvd: 0, sent: 0
IPv4 dropped (too large): 0
IPv4 dropped (no route): 0
IPv6 dropped (no route): 0
IPv4 dropped (transmission failed): 0
IPv6 dropped (transmission failed): 0
-----
IPv4-IPv6 NAT module unloaded.

```

Note: from the early demo perspective, it is also possible to compile and run Linux kernel with ip46nat module on a PC. For a details regarding module compilation, see section 5.

5.2 ip46nat: IPv4 to IPv6 NAT operation

During module insertion, v4addr parameter is specified. It is being used as a source IPv4 address filter, used with /24 bitmask. For example, if v4addr is equal to 192.168.1.0, all incoming traffic from 192.168.1.0/24 network will be translated to IPv6. Source IPv6 address will be created as a concatenation of the P prefix (v6prefixp parameter specified during module insertion) and IPv4 address. Destination IPv6 address will be created as a concatenation of the M prefix (v6prefixm parameter specified during module insertion). TTL field will be copied and decreased by 1.

For converted IPv6 packet to be transmitted successfully, IPv6 forwarding must be enabled. IPv6 routing must also be configured. See section 4 for details.

Please note that L4 layer (TCP, UDP, ICMP, etc.) checksums are not modified in any kind. That means that After IPv4 to IPv6 conversion, packets will not be accepted by destination router. They must be converted back to IPv4. That should not pose any concerns, however, as routers are not supposed to investigate L4 content at all. Thus packets must be converted back to IPv4 before they reach their destination.

Matched packet information will be reported in the following manner:

```
#IPv4 rcvd (rcvd so far: 3) [src=192.168.1.100, dst=192.168.1.1,looking for 192.168.1.0/24] *
```

5.3 ip46nat: IPv6 to IPv4 NAT operation

Once IPv4 packets are sent as IPv6, it is expected to receive responses. They are to be received as a IPv6 packets. Source IPv6 address will belong to the M prefix (v6prefixm parameter used during kernel module insertion) and will contain source IPv4 address embedded on 4 least significant octets. Destination IPv6 address will belong to the P prefix (v6prefixp parameter used during kernel module insertion) and will contain destination IPv4 address embedded on 4 least significant octets. If incoming IPv6 packet meets those criteria, it will be converted to IPv4 packet. Header checksum will be calculated, TTL will be decreased and packet will be sent.

For converted IPv4 packet to be transmitted successfully, IPv4 forwarding must be enabled. IPv4 routing must also be configured. See section 4 for details.

5.4 Firewall – Limiting extra traffic

Internally IPv4-to-IPv6 NAT works as an extra packet handler. It means that each incoming packet

may receive extra handling, i.e. conversion to IPv6 or IPv4. Regardless of the outcome (process matches specified criteria or not), it is still being processed by the kernel in a normal way. For example, when criteria matching IPv4 arrives, there will be actually 2 new packets transmitted: first - IPv4 packet according to normal routing, and second – IPv6 packet generated by the ip4nat module. To avoid this behavior, iptables filtering may be used. See section 7.1 for example.

5.5 IPv4 over IPv6 tunneling

After ip6_tunnel module with all required dependencies is loaded, it is possible to configure IPv4-over-IPv6 tunnel using ip command that is part of the iproute suite. As Ipv4-over-IPv6 tunneling is a very new addition, it required fairly recent version of the iproute code. See Links section for appropriate download links. Make sure to use version that is dedicated to the kernel that is being used.

After kernel is booted and appropriate ip command is available, it is possible to configure IPv4-over-IPv6 tunnel. Assuming that local node has address 2000::1 and remote tunnel endpoint is 2000::abcd, tunnel can be created using following commands:

```
ip -6 tunnel add foo mode ipip6 local 2000::2 remote 2000::100
ip link set up foo
```

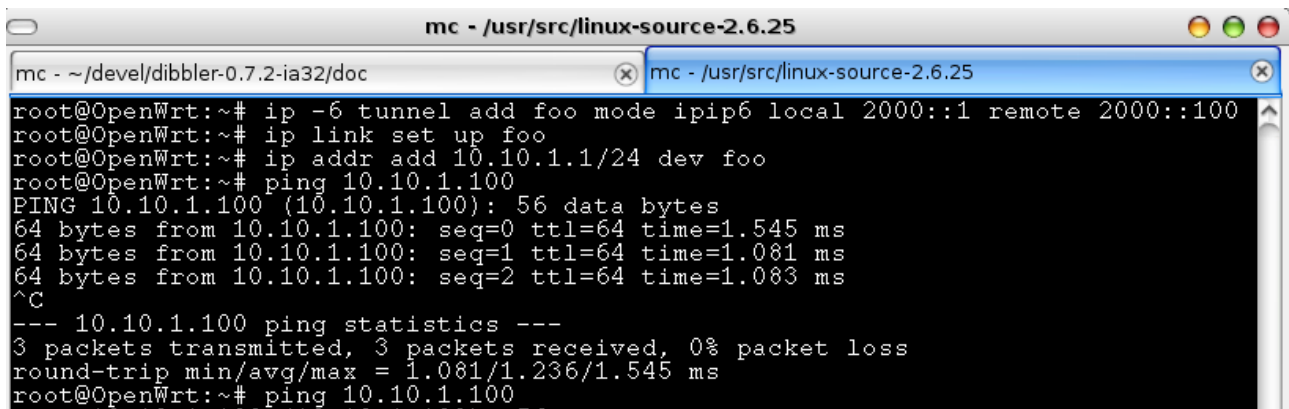
foo is a unique interface name that will be associated with this new tunnel. After tunnel is created, it is possible to add IPv4 address to local endpoint. If IPv4 address is also added to the remote end, IPv4 traffic over IPv6 is possible:

```
ip -6 tunnel add foo mode ipip6 local 2000::1 remote 2000::100
ip link set up foo
```

After that operation, IPv4 addresses may be assigned to the tunnel. If both tunnel endpoints – local (on this node) and remote (on the remote node) have assigned IPv4 addresses, normal IPv4 traffic may be exchanged between them. Routing may be configured to transmit IPv4 traffic via such interface:

```
ip route del default
ip route add default dev foo
```

For example tunnel configuration, see Fig. 5 below.



```
mc - /usr/src/linux-source-2.6.25
mc - ~/devel/dibbler-0.7.2-ia32/doc
root@OpenWrt:~# ip -6 tunnel add foo mode ipip6 local 2000::1 remote 2000::100
root@OpenWrt:~# ip link set up foo
root@OpenWrt:~# ip addr add 10.10.1.1/24 dev foo
root@OpenWrt:~# ping 10.10.1.100
PING 10.10.1.100 (10.10.1.100): 56 data bytes
64 bytes from 10.10.1.100: seq=0 ttl=64 time=1.545 ms
64 bytes from 10.10.1.100: seq=1 ttl=64 time=1.081 ms
64 bytes from 10.10.1.100: seq=2 ttl=64 time=1.083 ms
^C
--- 10.10.1.100 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.081/1.236/1.545 ms
root@OpenWrt:~# ping 10.10.1.100
```

Fig. 5: Tunnel creation

5.6 DNS proxy

IPv4 based DNS queries from the client can be handled in the same way as any other traffic, i.e. translated to IPv6 by using ip46nat module or tunneled over IPv6 using ip6_tunnel. However, it is also possible to use LinkSys device as a DNS proxy. It will receive incoming IPv4 queries from the client, issue IPv6 based queries to a specified ISP DNS server, handle returning IPv6 based responses and send its contents as a IPv4 based reply to the client. To set up DNS proxy, dnsmasq software may be used. To install it, issue following command:

```
ipkg install dnsmasq_2.45-1_mipsel.ipk
```

After the software is installed, it starts dnsmasq by default. It should be stopped:

```
/etc/init.d/dnsmasq
```

Assuming client's segment (br-lan interface) uses 192.168.1.0/24 pool and the ISP's DNS server is available at the abcd::1 address (eth0.1 interface), following command may be used:

```
dnsmasq -b -d -i br-lan -R --server=2000::1
```

The meaning of the switches is as follows:

- -b – Fake reverse lookups for RFC1918 private IPv4 addresses
- -d – Don't run as a daemon. In the initial runs, it is better to run this command in the console, so any potential issues generated by this app will be discovered.
- -i br-lan – Accept queries received on the br-lan interface
- -R – Ignore /etc/resolv.conf. Without this option, dnsmasq will also use all DNS entries from the /etc/resolv.conf file. To prevent this, -R option should be used.
- --server=2000::1 Forward all queries to the 2000::1 server

For the complete list of switches, please use dnsmasq -help command or see its man page:

<http://www.thekelleys.org.uk/dnsmasq/docs/dnsmasq-man.html> Complete list is also available at the project website: <http://klub.com.pl/ip46nat/snapshots/2008-10-03/dns-options.txt>

Example dnsmasq execution is presented below:

```
root@OpenWrt:~# dnsmasq -b -d -i br-lan -R --server=2000::1
dnsmasq: started, version 2.45 cachesize 150
```

```
dnsmasq: compile time options: IPv6 GNU-getopt ISC-leasefile no-DBus no-
I18N TFTP
dnsmasq: using nameserver 2000::1#53
dnsmasq: read /etc/hosts - 1 addresses
```

5.7 Router Advertisements daemon

Although not strictly related to this project goals (it is assumed that client's network is purely IPv4), it is possible to provide support for IPv6 in the client's network. Router Advertisement daemon (radvd) should be used for this. To install it, use following command:

```
ipkg install radvd_1.1-1_mipsel.ipk
```

Make sure that /etc/radvd.conf file contains proper configuration. Also radvd will refuse to start if IPv6 packet forwarding is disabled.

6 Remote configuration via DHCPv6

This section provides description how to achieve remote configuration of various aspects of the linksys device configuration, e.g. traffic type or used addresses.

6.1 Dibbler installation

Note: During phase 1, dibbler 0.7.1 was used. Phase 2 now uses 0.7.2 that covers both functionalities: IPv4-to-IPv6 NAT and IPv4-over-IPv6 tunneling.

To install dibbler client, copy dibbler-client_0.7.2-1_mipsel.ipk, dibbler-server_0.7.2-1_mipsel.ipk and uclibcxx_0.2.2-1_mipsel.ipk files to /tmp directory on the OpenWRT box (please use newer versions, if available):

```
scp *.ipk root@192.168.1.1:/tmp
```

Make sure that your PC is in the same network as OpenWRT box. After packages are copied, install them using ipkg install command on the OpenWRT box. Make sure that uclibcxx package is installed before dibbler packages (both dibbler packages require uclibcxx package to work). Dibbler software may be installed with the following set of commands:

```
cd /tmp
ipkg install uclibcxx_0.2.2-1_mipsel.ipk
ipkg install dibbler-client_0.7.2-1_mipsel.ipk
ipkg install dibbler-server_0.7.2-1_mipsel.ipk
```

See Fig. 6 for example dibbler client installation process.


```
Terminal
root@OpenWrt:/tmp# cat /etc/dibbler/client.conf
log-mode short
iface eth0 {
    pd
}
root@OpenWrt:/tmp# dibbler-client run
Dibbler - a portable DHCPv6, version 0.7.1 (CLIENT, Linux port)
Authors : Tomasz Mrugalski<thomson(at)klub.com.pl>, Marek Senderski<msend(at)o2.pl>
Licence : GNU GPL v2 or later. Developed at Gdansk University of Technology.
Homepage: http://klub.com.pl/dhcpv6/
2000.1.3 2:4:2 Client Notice My pid (663) is stored in /var/lib/dibbler/client.pid
2000.1.3 2:4:2 Client Notice Detected iface wlan0/6, MAC=0:90:4c:5f:0:2a.
2000.1.3 2:4:2 Client Notice Detected iface wmaster0/5, MAC=0:90:4c:5f:0:2a.
2000.1.3 2:4:2 Client Notice Detected iface eth0.1/4, MAC=0:1d:7e:bc:41:2a.
2000.1.3 2:4:2 Client Notice Detected iface eth0.0/3, MAC=0:1d:7e:bc:41:2a.
2000.1.3 2:4:2 Client Notice Detected iface eth0/2, MAC=0:1d:7e:bc:41:2a.
2000.1.3 2:4:2 Client Notice Detected iface lo/1, MAC=0:0:0:0:0:0.
2000.1.3 2:4:2 Client Notice Parsing /etc/dibbler/client.conf config file...
4:2 Client Debug Prefix delegation option found.
4:2 Client Debug Parsing /etc/dibbler/client.conf done, result=0
4:2 Client Debug 1 interface(s) specified in /etc/dibbler/client.conf
4:2 Client Info Interface eth0/2 configuration has been loaded.
4:2 Client Info My DUID is 0:1:0:1:0:2:a9:69:0:1d:7e:bc:41:2a.
4:2 Client Debug Bind reuse enabled (multiple instances allowed).
4:2 Client Notice Creating control (:) socket on the lo/1 interface.
4:2 Client Notice Creating socket (addr=fe80::21d:7eff:febc:412a) on the eth0/2 interface.
4:2 Client Info Socket bound to fe80::21d:7eff:febc:412a/port=546
4:2 Client Info Creating SOLICIT message with 0 IA(s), no TA and 1 PD(s) on eth0/2 interface.
4:2 Client Debug Sleeping for 1 second(s).
```

Fig. 7: Client start

6.2 Using dibbler to control NAT (obsolete)

Note: This approach is considered obsolete and was superseded by more universal tunnel-mode. See following sections for details.

It is possible to use dibbler-client to control IPv4-to-IPv6 NAT. To do so, dibbler-client must be switched to a special mode. Instead of handling prefixes in a usual manner, it will call special external scripts instead. Those shell scripts may be easily modified to fulfill specific purposes. There are two scripts: `mappingprefixadd` and `mappingprefixdel`. Both must be stored in the `/var/lib/dibbler` directory. During execution, every script will get exactly 1 parameter that will be the (added or deleted) prefix. Here is the example of the configuration file.

```
log-mode short
experimental
mapping-prefix
iface eth0.0 {
    pd
}
```

Experimental keyword instructs dibbler to allow use experimental (i.e. non-standard) code. One of such non-standard extensions is `mapping-prefix`. Those scripts are the best place to add fixed parameters.

6.3 Remote traffic type control via DHCPv6 (Tunnel-mode)

To remotely configure tunnel or a NAT, extra DHCPv6 options have been defined. Implementation of those options have been provided in the Dibbler implementation. They are encapsulated in one vendor-specific information option.

Following suboptions have been defined:

- Tunnel Type (option code=1,length=1 byte): specifies what kind of traffic encapsulation should be used:

prefix, tunnel endpoint 2000:4f8::1 and tunnel mode 1, execution will take following form:

```
./notify 2000::1 2000:abc:: 64 2000:4f8::1 1 add
```

6. Notify script will act depending on the received parameters. It will load necessary modules, create and setup tunnels, configure routing and perform other required tasks.
7. After T1 timeout, client will initiate RENEW message exchange. After receiving REPLY, client will again run the script with all updated parameters:

```
./notify 2000::1 2000:abc:: 64 2000:4f8::1 1 modify
```

8. Although in normal operation graceful shutdown will never happen, when being shut down, client also calls the script with delete parametr:

```
./notify 2000::1 2000:abc:: 64 2000:4f8::1 1 delete
```

6.5 notify script

As explained in previous section, notify script performs vital role in the remote tunnel configuration. That script along with now obsolete scripts used in phase 1, is available at the project website. It is shell script designed to be working under ash shell. OpenWRT uses ash instead of bash. See section 10 for details. Legacy scripts (`mappingprefixadd` and `mappingprefixdel`) are provided just for backward compatibility. Those scripts are to be placed in the `/var/lib/dibbler` directory.

6.6 Possible enhancements

It is worth noting that, although currently there are 2 tunnel types specified, it is very easy to add extra types. It is just matter of modifying configuration file (`server.conf`) to use new type and modify shell script only (`/var/lib/dibbler/notify`). No dibbler recompilation is required. Also, it is possible to use standard parameters like delegated prefix or assigned IPv6 address for some extra operation (both values are passed as script parameters).

In case of extensive modification, e.g. when 2 prefixes are delegated to one device, it is also possible to monitor actual set of parameters assigned to the device. Dibbler client stores all assigned addresses and prefixes in the `/var/lib/dibbler/client-AddrMgr.xml` file. All extra parameters are stored in `/var/lib/dibbler/client-IfaceMgr.xml`. One or both files may be processed in the `notify` script execution.

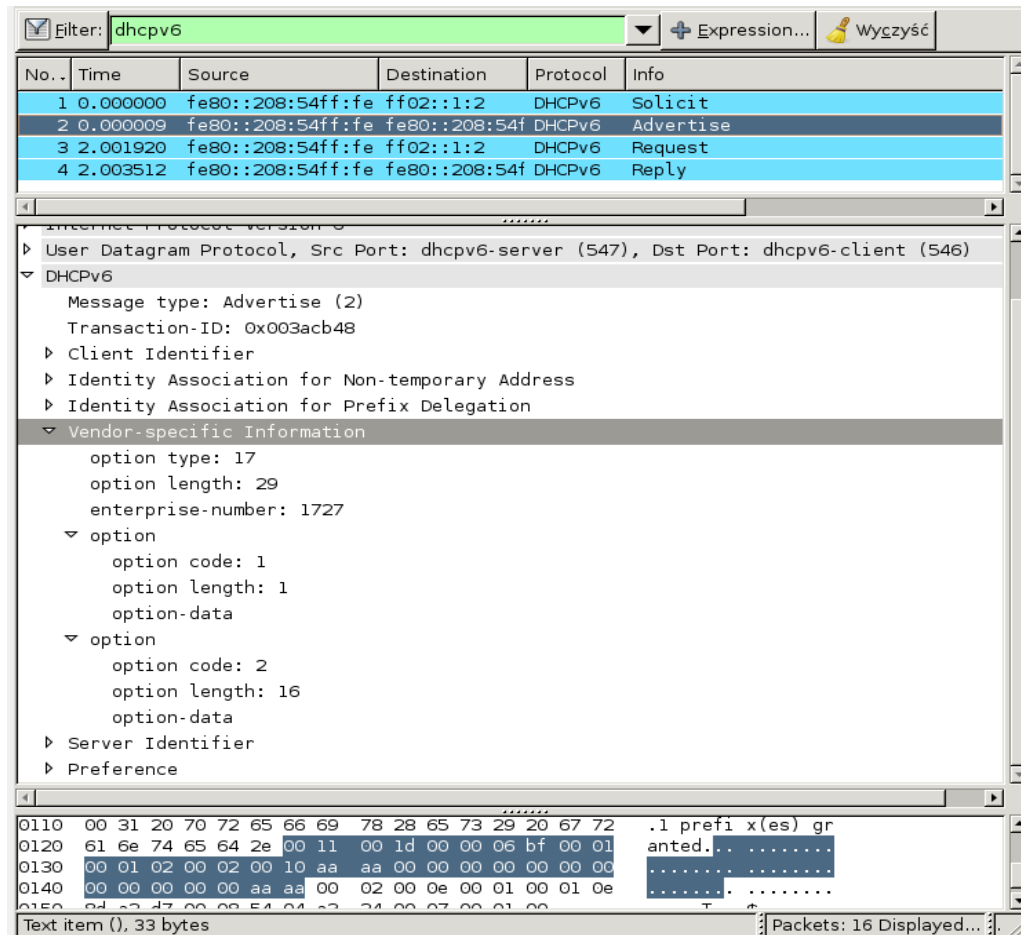


Fig. 8: Tunnel-mode transported as vendor-specific option

6.7 Client configuration

Client should be configured to ask for 1 address, 1 prefix, tunnel-mode and possibly other configuration parameters, e.g. DNS servers. Such configuration can be requested by using following client.conf:

```

log-mode short
log-level 8

experimental
tunnel-mode 1727
notify-scripts
iface eth0
{
  ia
  pd
}
  
```

6.8 Server configuration

To properly configure client, server must be able to provide vendor-specific information option with several suboptions. Simple server configuration that is able to provide tunnel-mode options is presented below. Server will send tunnel-mode information as vendor-specific information option with vendor-id = 1727. It will be set tunnel-type to 1 (IPv4-over-IPv6 NAT) and remote tunnel

endpoint will be 2000::100. Clients will receive addresses from the 2000::/120 pool and 64bit long prefixes will be delegated from the 3000::/48 address space.

```
log-level 8
log-mode short

experimental
tunnel-mode 1727 2 2000::100
iface "eth0"
{
    class
    {
        pool 2000::/120
    }
    pd-class
    {
        pd-pool 3000::/48
        pd-length 64
    }
    option dns-server 2000::ff
    option domain example.com
}
```

6.9 Client-specific server configuration

Scenario above assumes that every client's tunnel has the same remote endpoint. That approach is easier from the maintenance point of view, but it scales poorly (too much traffic running thru one machine). Therefore in some cases it may be better to spread the traffic between multiple nodes, so different tunnel endpoints will be used. Server can also be configured to provide different parameters for each client. That can be arranged using exceptions. In the following example, client with DUID 00:01:00:01:0f:54:92:72:00:08 will be ordered to create tunnel type 2 (IPv4-over-IPv6) to the 2000::aaaa address.

Although RFC3315 recommends that clients should used DUID-LLT (DHCP Identifier based on link address and timestamp), it may be more convenient to use DUID-LL (DHCP Identifier based on link address only). That has huge benefit of being completely predictable. There are 2 ways of forcing client to use specific DUID:

1. Use “duid-type duid-ll” in the client.conf file. Dibbler client will take this into consideration during first execution when DUID is created.
2. Store desired DUID in the /var/lib/dibbler/client-duid file. Any properly formed values may be used.

```
log-level 8
log-mode short

experimental
tunnel-mode 1727 1 2000::100
iface "eth0" {
    class {
        pool 2000::/123}
    }
    pd-class {
        pd-pool 3000::/48
        pd-length 64
    }
}
```

```
option dns-server 2000::ff
option domain example.com

client duid 0x000100010f5492720008
{
    tunnel-mode 1727 2 2000::aaaa
}

client duid 0x000100010f549272000822222221
{
    tunnel-mode 1727 2 2001::eeee
}
```

7 Compilation

All parts of the ip46net project and its associated software is distributed as an open source. Therefore full source code is available and can be compiled.

7.1 Linux for embedded devices (OpenWRT)

ip46nat project uses development branch (“kamikaze”) of the embedded Linux distribution called OpenWRT. First step to build a firmware for your embedded device is to download OpenWRT. OpenWRT is a set of tools that automate building process of the firmware. There are several ways of obtaining sources. It is possible to download stable sources or use SVN repository instead. For the stability purposes, all development related to ip46nat is done on one specific SVN snapshot, revision 12386. (During phase 1 development, revision 11276 was used, but it didn't support 2.6.25 kernel that was required for features used in phase 2). To obtain this revision, issue following command:

```
svn co -r 12386 https://svn.openwrt.org/openwrt/trunk/
```

After checkout is complete, initial configuration takes place. To start configuration, type:

```
make menuconfig
```

Please note that, although using the same framework, that interface is significantly different from a similarly looking, Linux kernel configuration. Also, you may want to postpone this step and install additional patches as described in the following sections. Example of the OpenWRT configuration process is presented in Fig. 6.

Alternatively, sources with necessary patches are available on the ip46nat project website. You can download and extract them instead of getting the source code from the OpenWRT's SVN repository.

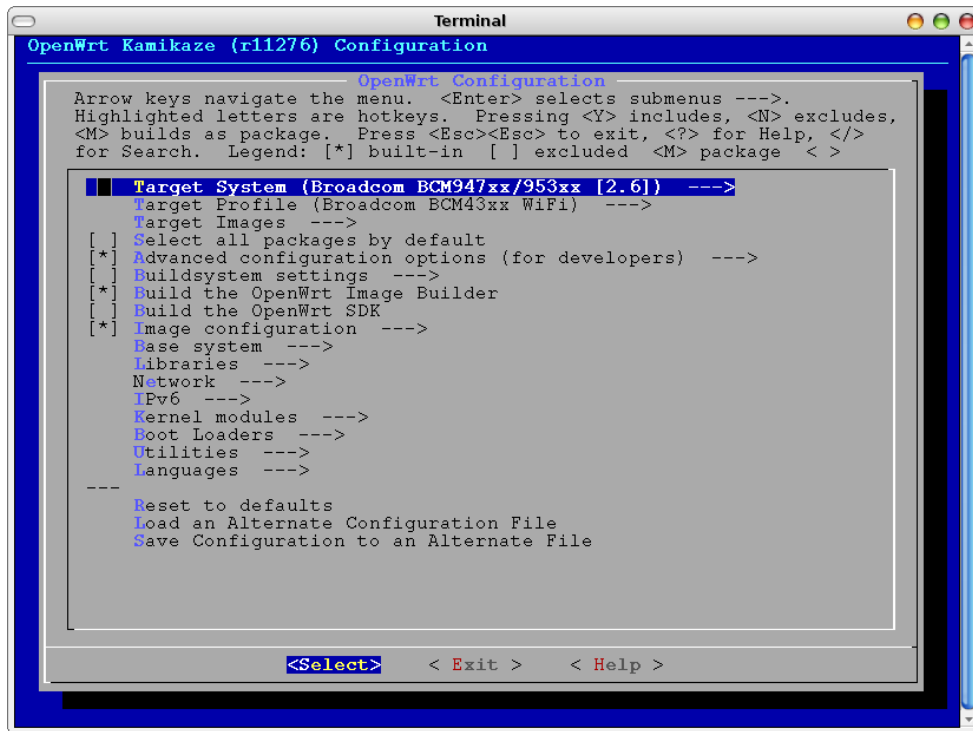


Fig. 9: OpenWRT configuration example

After configuration is complete, type `make` to download requires source code, build tool chain required for cross-compilation, all target binaries and images. This may take several hours, depending on the speed of the network connectivity and CPU. After subsequent rebuilds, this process is much shorter. For extra verbosity level, `make V=99` command may be used.

Make sure that the PC has Internet connectivity as OpenWRT downloads multiple additional packages, like kernel source. After compilation is complete, all packages and firmware images are available in the `bin/` directory.

7.2 ip46nat kernel module

`ip46nat` is a Linux kernel module. Although developed with embedded environment in mind, it is not specific for embedded devices. It may be run on any device that runs Linux kernel. That includes ordinary PC boxes. Due to extensive set of debugging and tracking tools available, it may be beneficial to run this module on a PC, at least during early configuration stages.

`Ip46nat` source code is available as a patch for various Linux kernels. It is recommended to use only those kernels that patches are specifically provided for. In general, it is likely that patch prepared for a specific kernel version will work fine on other, similar version. But it may also fail.

To compile `ip46nat` as a module on a Linux box, follow this steps:

1. Download supported Linux kernel. This example assumes that Linux kernel 2.6.25.16 will be used.
2. Extract kernel using command: `tar xjvf linux-2.6.25.16.tar.bz2`
3. Apply `ip46nat` patch: `patch -p0 < ip46nat-2.6.25.16.patch`
4. Setup kernel configuration in a normal way: `make menuconfig`
5. Go to Networking => Networking Options => IPv4-IPv6 NAT and select it as a module. If this option is not available, make sure that patch was applied properly and that networking

support, IPv4 and IPv6 are enabled.

6. Save kernel configuration (.config file)
7. Build kernel using following command: **make**
8. Build kernel modules using following command: **make modules**
9. Continue with normal kernel installation. See numerous installation help documents available here: <http://www.google.com/search?q=linux+kernel+installation>

7.3 ip46nat kernel module as an ipk package

To make ip46nat available from the OpenWRT configuration menu, download selected OpenWRT source (see section 7.1) and apply ip46nat-openWRT patch:

```
patch -p0 < ip46nat-openwrt-12386.patch
```

After successful patching, ip46nat module is available in the Kernel modules => Network support => kmod-ip46nat. Select it as a module. See Fig. 10.

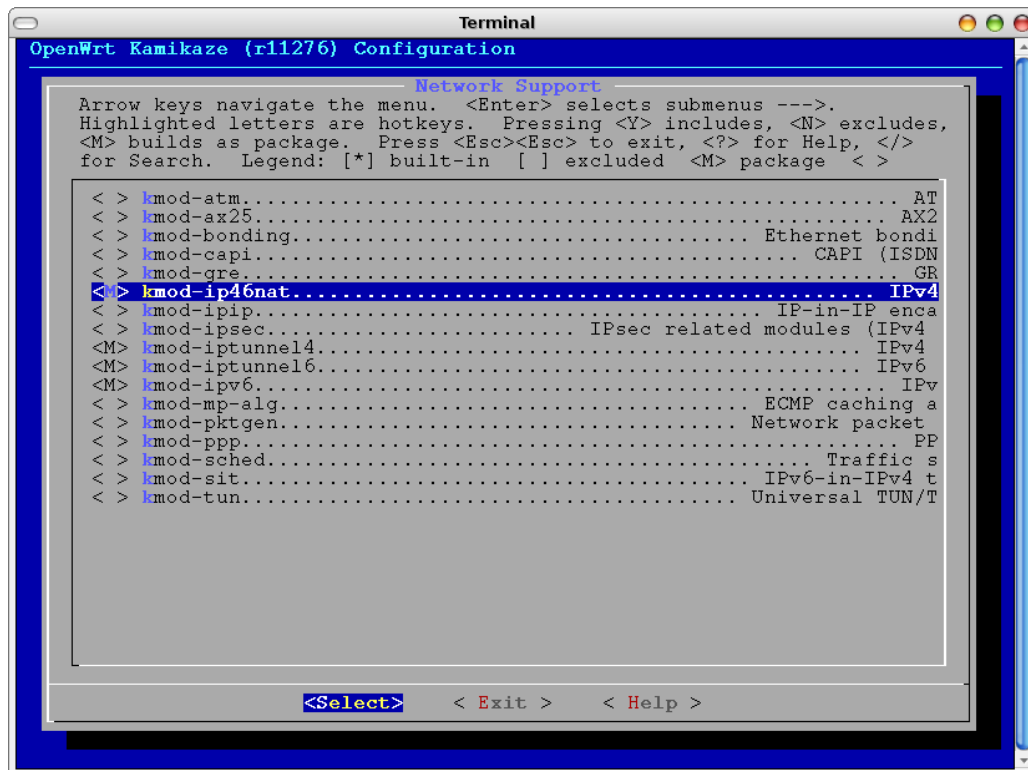


Fig. 10: ip46nat being selected in OpenWRT configuration menu

7.4 dibbler software

Dibbler software may be compiled for various systems: Windows, Linux or even embedded environment. In this section, instructions regarding Linux compilation are provided. For details regarding OpenWRT cross-compilation, please see next section.

Details regarding dibbler compilation are described in the Dibbler User's Guide, available on the project website: <http://klub.com.pl/dhcpv6/>. Advanced topics are also discussed in Dibbler Developer's Guide, also available on the same website.

To compile dibbler, download and extract latest sources. Dibbler consists of several entities: server, client, relay and requestor. To compile one or more components, issue make command followed by

name of the component. For example, to compile server, client and relay, following commands may be used:

```
tar zxvf dibbler-0.7.2-src.tar.gz
cd dibbler-0.7.2
make server client relay
```

After compilation is complete, dibbler-server, dibbler-client and dibbler-relay binaries will be available in the current directory.

Normally, dibbler is released under GNU GPLv2 or later license. However, due to some legal concerns regarding interpretation of the “or later” part, dibbler 0.7.2 was released under GNU GPLv2 only.

7.5 *dibbler software as an ipk package*

To compile dibbler for OpenWRT environment, several preparatory steps are necessary.

1. Checkout packages repository from the OpenWRT project:
svn co <https://svn.openwrt.org/openwrt/packages/>
Note that there may be an old dibbler package. Do not use it as it is broken.
2. Copy or symlink packages/libs/uclibc++ directory (from packages repository) to packages/uclibc++ directory (in the OpenWRT repository).
3. Download and extract openwrt-dibbler-0.7.2.tar.gz file.

Note: Those steps are not necessary when using source tree provided on the ip46nat project website.

After those steps are complete, go to the OpenWRT directory. There should be following directories:

```
docs/
include/
package/
scripts/
target/
toolchain/
tools/
```

and some additional files. In the package directory, there should be (among others) dibbler and uclibc++ directories.

To compile uclibc++ and dibbler packages, please follow instructions from section 7.1. In the OpenWRT configuration menu, go to Libraries => uclibcxx for uClibc++ and IPv6 => dibbler-server, dibbler-client and dibbler-relay for dibbler packages.

7.6 *IPv4-over-IPv6 tunnel modules*

Kernel 2.6.25 supports IPv4 over IPv6 tunneling. To compile this kernel with module supporting IPv4 over IPv6 tunneling, please download and extract 2.6.25 Linux kernel sources. Following command may be used for compilation preparation:

```
make menuconfig
```

Go to Networking menu, then Networking Options and select IPv6: IP-in-IPv6 tunnel. Also select all other required modules. After saving changes, type make to begin kernel compilation. There are numerous tutorials and walkthroughs available on the Internet regarding Linux kernel configuration.

Two modules will be created: ip6_tunnel.ko and tunnel6.ko. After booting up the kernel, those modules may be loaded using following command:

```
modprobe ip6_tunnel
```

Compiled modules are supposed to be used in the kernel they were compiled with. They may refuse to work with other kernel versions or even with kernels compiled from the same source version, but with different parameters.

8 IPv4-to-IPv6 NAT (Phase 1) testing

This section provides report from the phase 1 validation. It may also serve as a proof of concept. For the testing purposes, LinkSys will be referred to as DUT – Device Under Test. Most tests were performed in a configuration, where 2 PCs were connected via DUT. This scenario is presented in Fig. 11 below.

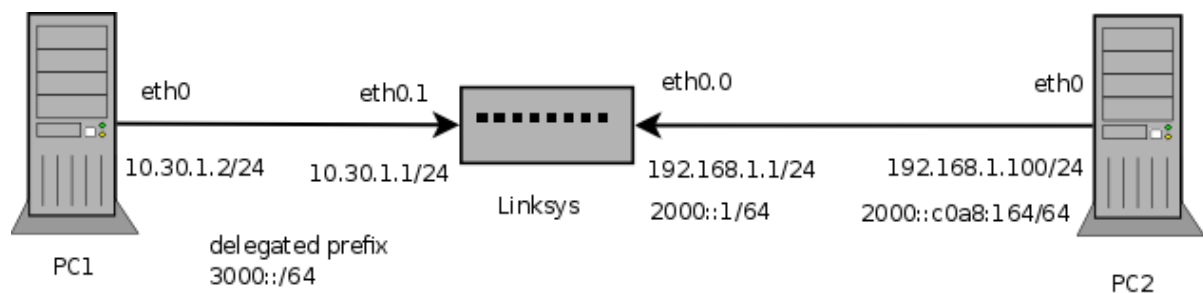


Fig. 11: Test network

8.1 IPv4 to IPv6 traffic

IPv4 packets were transmitted from the PC1. Single ICMPv4 packets were sent. Fig. 12 contains network captures of such packets. IPv4 packets after translation (they are IPv6 packets) are presented in Fig. 13. Please note that those IPv6 packets still carry ICMPv4 protocol data.

As packets size after translation increase, there is a size limit of the maximum IPv4 packets that may be translated. Translation adds extra 20 bytes, so maximum packet size is 1480. To transmit such packets, following command may be used:

```
ping -s 1452 192.168.1.100
```

Note that ping command uses -s (size) parameter to specify ICMPv4 protocol payload. That is increased by ICMP header (8 bytes) and IPv4 header (20 bytes). Therefore 1452+8+20 gives 1480. Any packets larger than this will be dropped by ip46nat module. To avoid such drops, the reasonable course of action is to limit MTU of the transmitting device. Also to help debugging such cases, ip46nat has dedicated statistics for too large IPv4 packet drops.

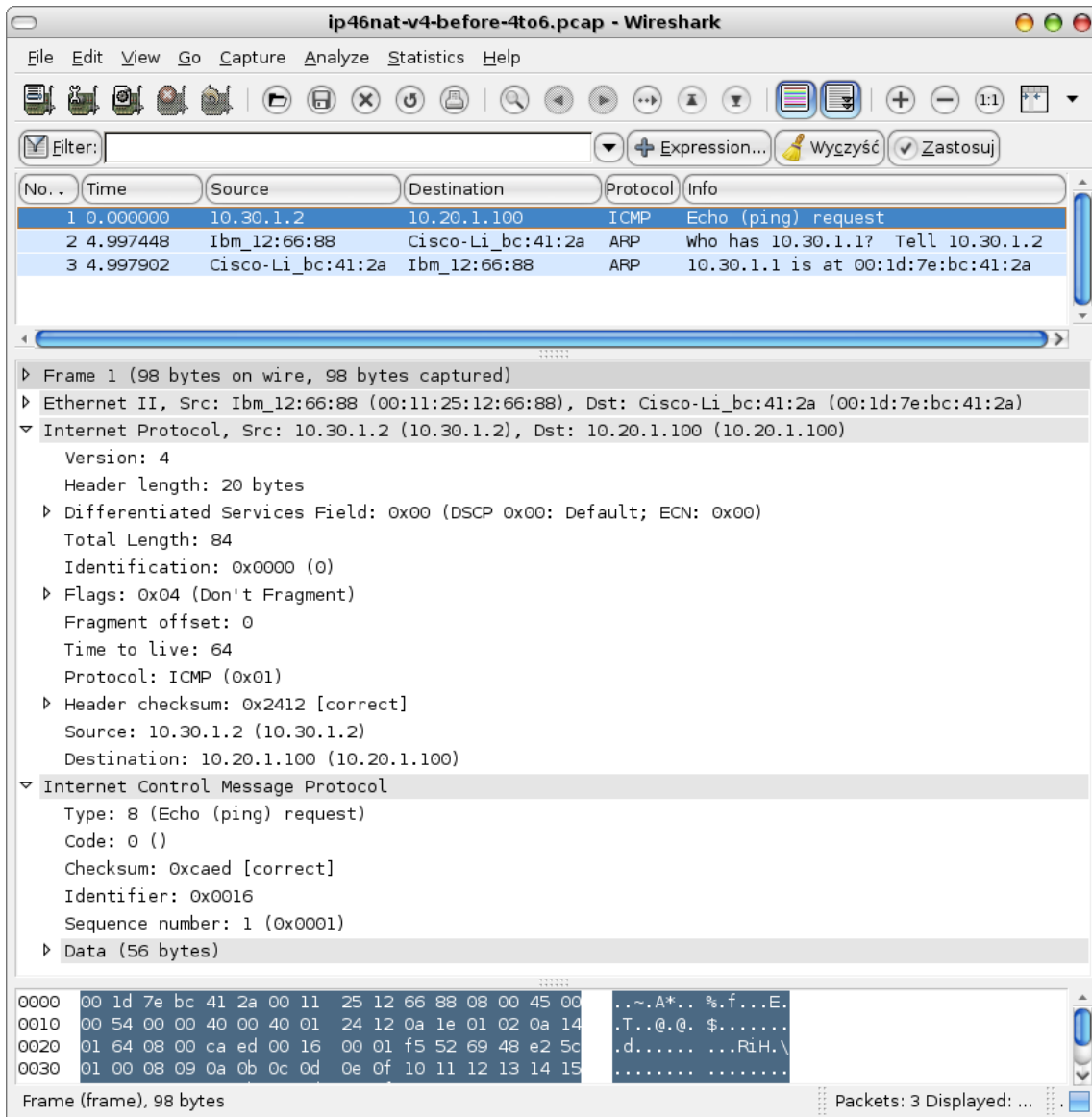


Fig. 12: IPv4 packet before translation

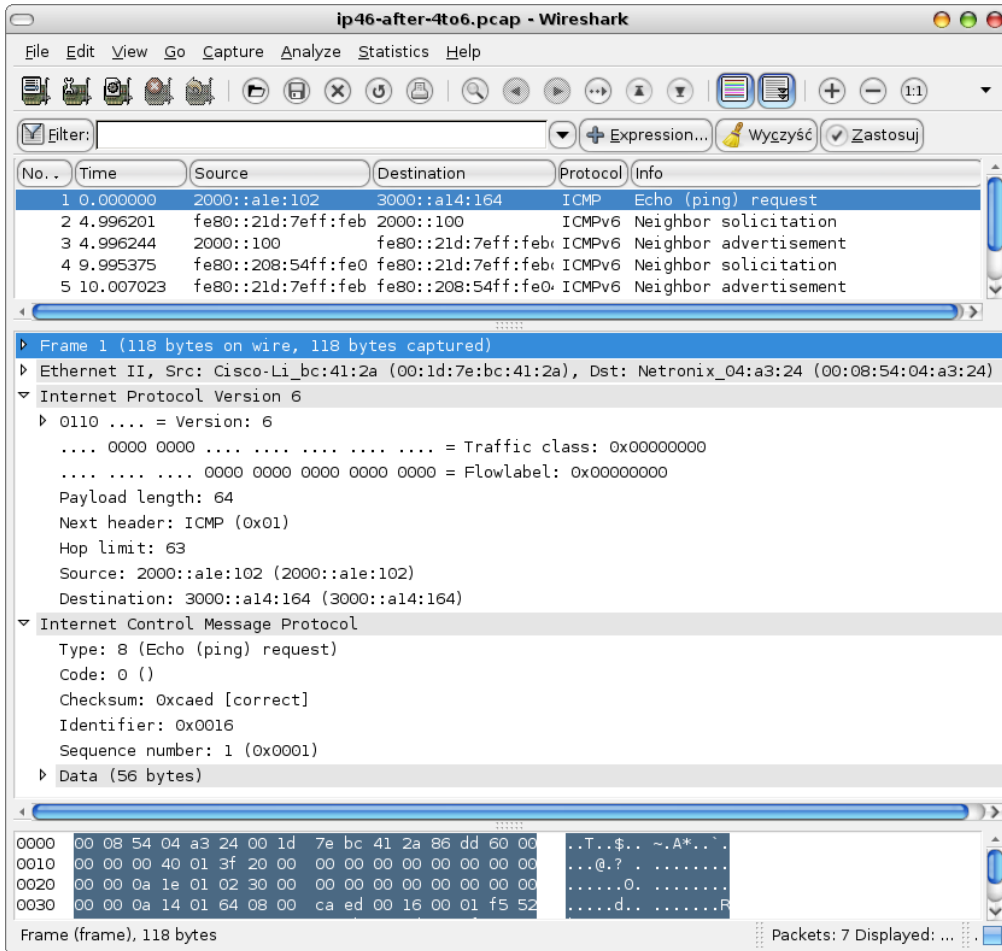


Fig. 13: IPv4 packets after IPv4-to-IPv6 translation

8.2 IPv6 to IPv4 traffic

Similar tests were performed for IPv6 traffic being translated to IPv4. See Fig. 14 for packets before translation and Fig. 15 for packets after translation.

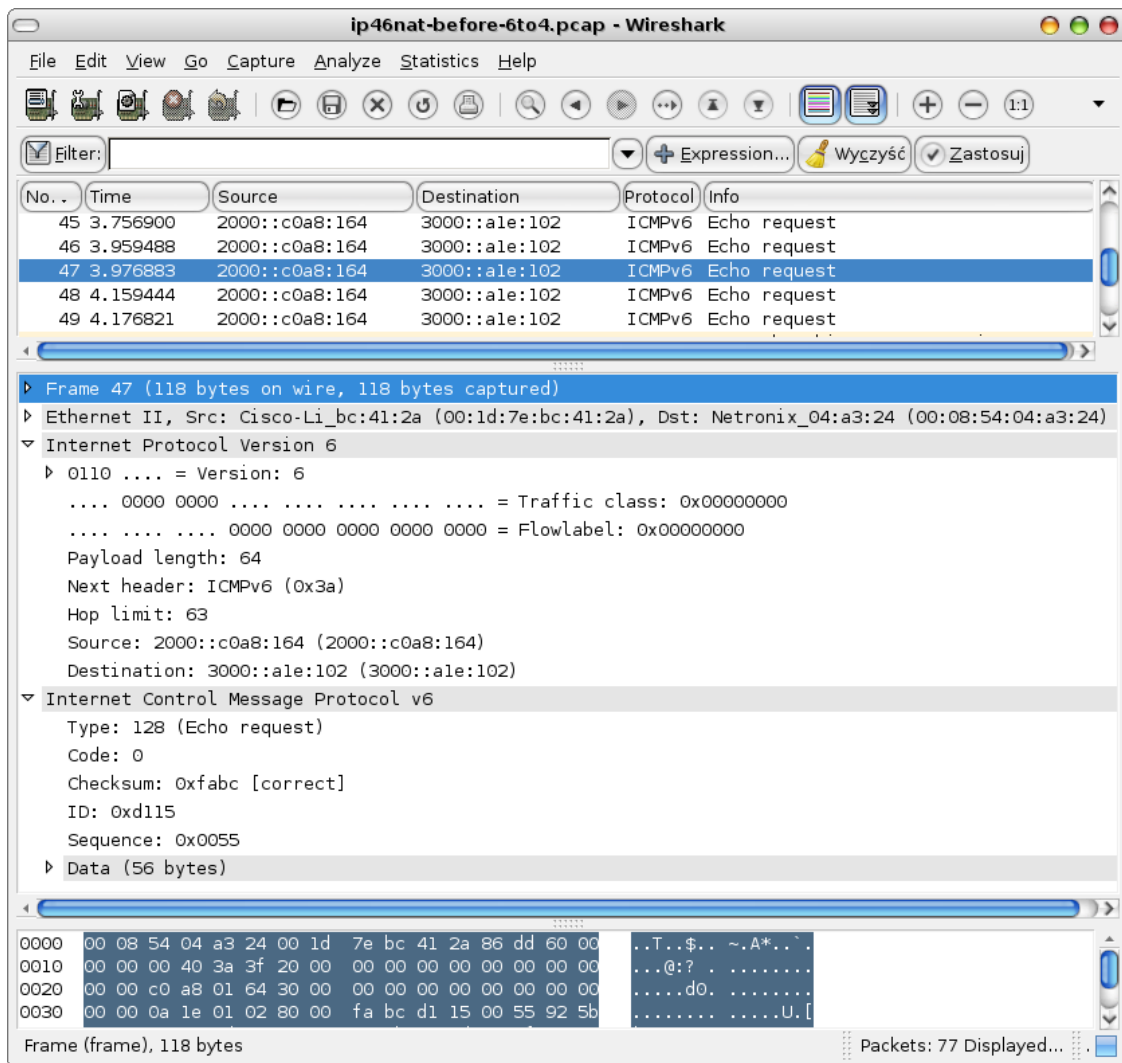


Fig. 14: IPv6 packet before reverse translation

Please note that to have packets transmitted, both its source and destination address should be legitimate, i.e. routing should be configured for such packets. For example, IPv6 packet `src=2000::1,dst=2000::c0a8:164` will be translated to IPv4 packet `src=0.0.0.1,dst=192.168.1.100`. Although destination address is valid, source address is not and thus ip46nat will not be able to find appropriate route and will drop the packet. This error case will be logged accordingly. There is also separate statistic for this condition.

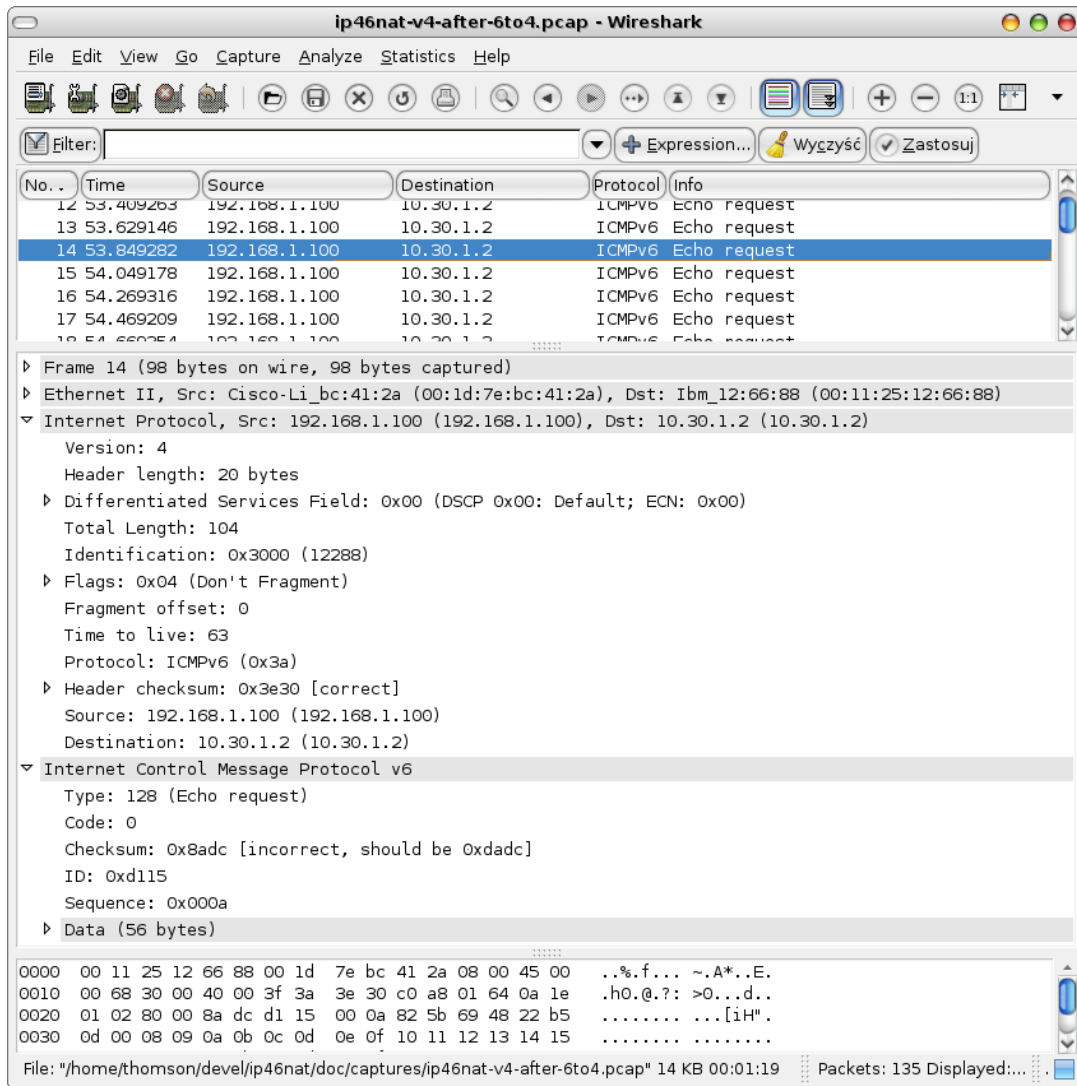


Fig. 15: IPv6 packet after reverse translation

8.3 Firewall configuration

Due to internal Linux kernel architecture, after ip46nat module processes the packet, it is not possible to forbid normal Linux IPv4 routing procedures to continue. To avoid packet duplication, IPv4 packets that are to be translated, should be dropped in a postprocessing phase (they will be sent as IPv6 only). Assuming network configuration as in Fig. 11 (end-user IPv4 segment is on the left side), following command may be used to drop packets AFTER being handled by ip46nat:

```
iptables -t nat -I POSTROUTING -s 10.30.1.0/24 -d ! 10.30.1.0/24 -j DROP
```

8.4 Negative testing

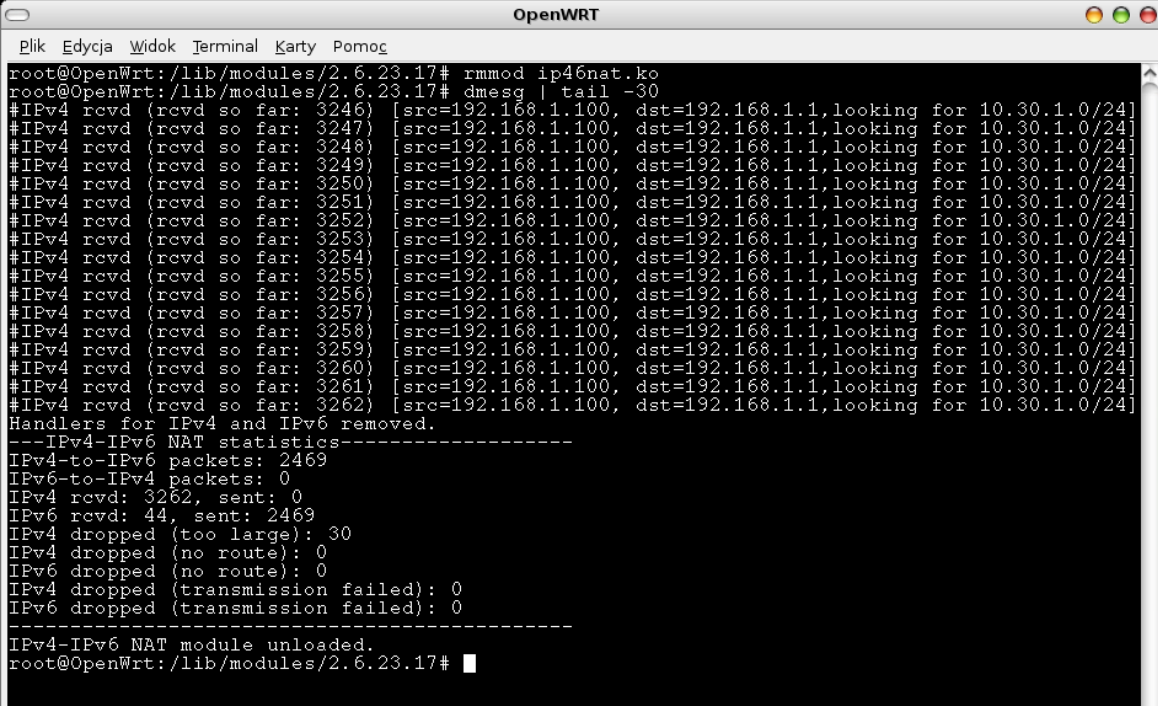
Too large IPv4 packets were sent to confirm that no buffer overflow occurs. Packets were dropped as expected and appropriate statistic was increased. Numerous not-matching criteria IPv4 and IPv6 packets were sent. None of them was ever translated. For easier debugging, ip46nat module prints every packet that is received, its source and destination address with values of configured filter. Packets matching criteria are labeled with asterisk (*).

8.5 Performance testing

To make sure that the device is being able to handle heavy traffic, biggest possible (1480) packets were transmitted in an continuous manner. After prolonged traffic handle the device behaved normally. See Fig. 14 for statistics after such testing.

8.6 Statistics

As specified in Fig.14, there are several statistics implemented. They may be used for overall operation measurements. They are printed after module is unloaded. To see them, use dmesg command. It seems useful to filter dmesg's output, e.g. Using tail -30 command.



```
OpenWRT
Plik Edycja Widok Terminal Karty Pomoc
root@OpenWrt:/lib/modules/2.6.23.17# rmmod ip4nat.ko
root@OpenWrt:/lib/modules/2.6.23.17# dmesg | tail -30
#IPv4 rcvd (rcvd so far: 3246) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3247) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3248) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3249) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3250) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3251) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3252) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3253) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3254) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3255) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3256) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3257) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3258) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3259) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3260) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3261) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
#IPv4 rcvd (rcvd so far: 3262) [src=192.168.1.100, dst=192.168.1.1, looking for 10.30.1.0/24]
Handlers for IPv4 and IPv6 removed.
-----IPv4-IPv6 NAT statistics-----
IPv4-to-IPv6 packets: 2469
IPv6-to-IPv4 packets: 0
IPv4 rcvd: 3262, sent: 0
IPv6 rcvd: 44, sent: 2469
IPv4 dropped (too large): 30
IPv4 dropped (no route): 0
IPv6 dropped (no route): 0
IPv4 dropped (transmission failed): 0
IPv6 dropped (transmission failed): 0
-----
IPv4-IPv6 NAT module unloaded.
root@OpenWrt:/lib/modules/2.6.23.17#
```

Fig. 16: Statistics after some traffic

8.7 Test conclusion

Developed software, while being remotely configurable via DHCPv6, is able to translate IPv4 to IPv6 efficiently. Reverse traffic is also handled properly. Although no end-user solution was provided, all building blocks for experienced user are provided, accompanied with documentation that explains how to achieve the ultimate goal by executing all intermediate steps.

9 IPv4 over IPv6 tunneling (Phase 2) testing

IPv4 over IPv6 validation can be split into 2 parts: traffic validation and remote configuration validation. For a realistic test environment, 2 PCs were connected to the DUT: PC1 is a end-user's equipment (Vista PC). PC2 is a operator's server (Linux PC). This architecture is presented in Fig. 17. PC2 also acts as a web server. It has configured IPv4-over-IPv6 tunnel. There is extra IPv4 address (192.168.2.100) assigned to the tunnel.

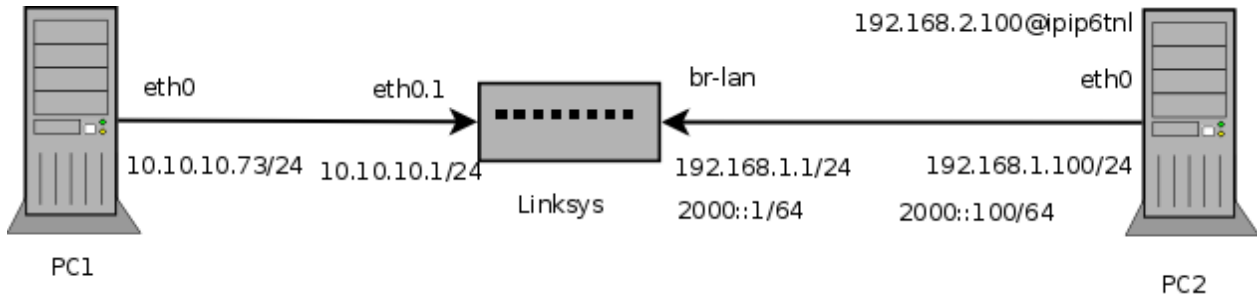


Fig. 17: IPv4 over IPv6 testing infrastructure

9.1 Traffic validation

Goal of this test was to validate if DUT is handling IPv4-over-IPv6 traffic properly. Tunnel on DUT was configured according to description in section 7.6.

It receives incoming IPv4 packets on the eth0.1 from the 10.10.10.0/24 class. That traffic is encapsulated in IPv6 packets and sent via br-lan interface. Returning IPv4 packets are received on the br-lan. To simulate real destination, PC2 has extra IPv4 address (192.168.2.100) assigned to the ipip6tnl interface (local name for the tunnel interface).

Tests started with simple ICMP messages. Pings were sent from the DUT to PC2. This exchange is presented in Fig. 18.

```

mc - /usr/src/linux-source-2.6.25
mc - ~/devel/dibbler-0.7.2-ia32/doc
root@OpenWrt:~# ip -6 tunnel add foo mode ipip6 local 2000::1 remote 2000::100
root@OpenWrt:~# ip link set up foo
root@OpenWrt:~# ip addr add 10.10.1.1/24 dev foo
root@OpenWrt:~# ping 10.10.1.100
PING 10.10.1.100 (10.10.1.100): 56 data bytes
64 bytes from 10.10.1.100: seq=0 ttl=64 time=1.545 ms
64 bytes from 10.10.1.100: seq=1 ttl=64 time=1.081 ms
64 bytes from 10.10.1.100: seq=2 ttl=64 time=1.083 ms
^C
--- 10.10.1.100 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.081/1.236/1.545 ms
root@OpenWrt:~# ping 10.10.1.100

```

Fig. 18: ping from DUT via IPv4 over IPv6

The same ICMP echo/reply interaction was captured using Wireshark tool. That is presented in Fig. 19 below.

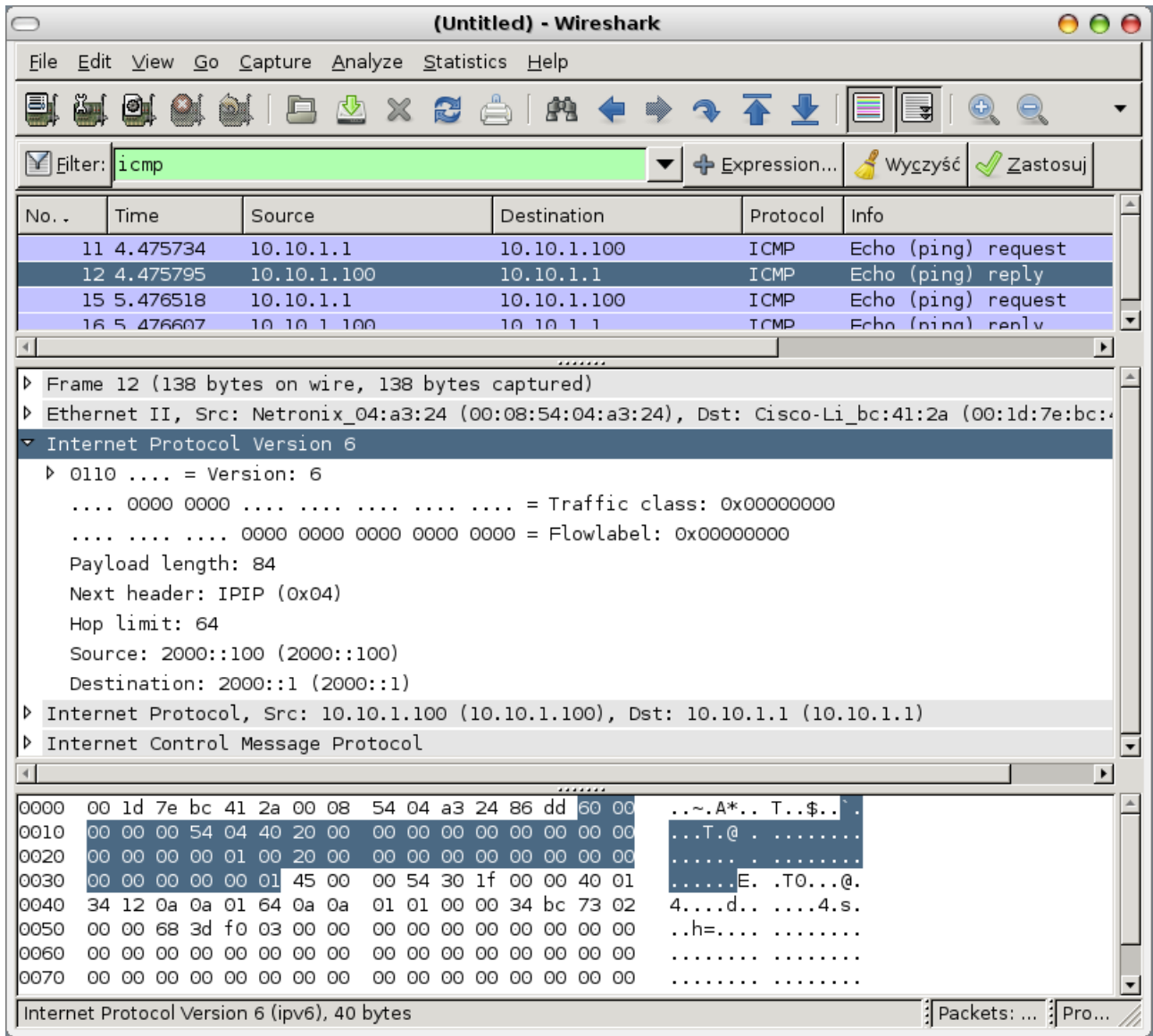


Fig. 19: ICMP transmission observed on Wireshark

After basic functionality was confirmed, web server was started on PC2. PC1 has been configured to use DUT as a default gateway. Also DUT has its default route configured via tunnel. It is possible that client (PC1, 10.10.10.73) is able to reach its destination (192.168.2.100), as confirmed with the ping interface. After starting web browser on the PC1, it was possible to connect to the PC2 and download web page content. This transmission was captured using Wireshark. That traffic is depicted in Fig. 20 below.

That scenario concluded traffic handling validation.

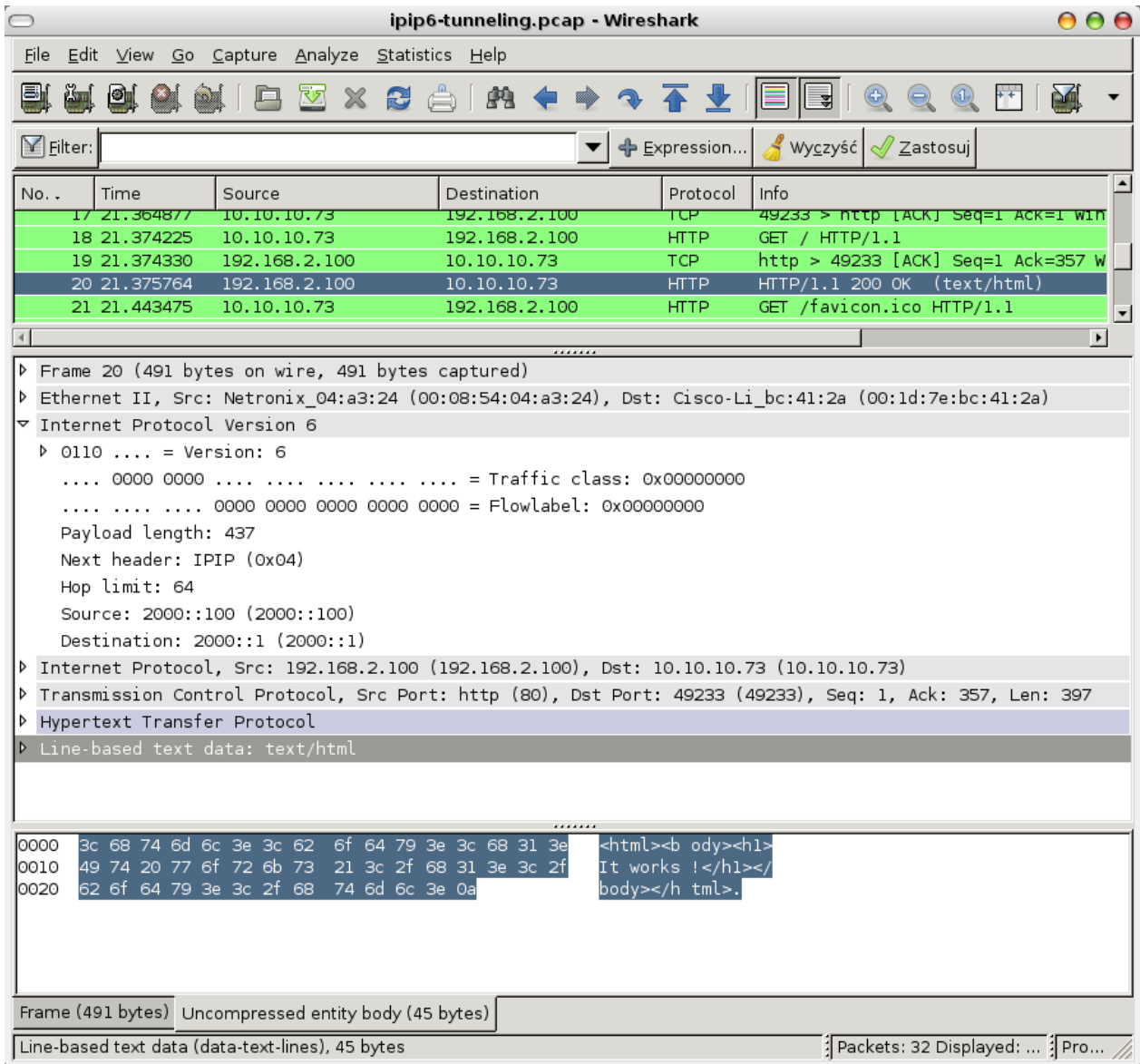


Fig. 20: HTTP traffic over IPv4 over IPv6

9.2 Remote ipip6 configuration validation

After traffic handling was confirmed, the next validation step was DHCPv6 operation. PC2 had dibbler-server installed. This server was configured to provide IPv6 addresses, prefixes and tunnel-mode options. Client running on DUT was configured to use the same vendor-id value as server. After initiation, client was able to find server and obtain necessary parameters via normal SOLICIT, ADVERTISE, REQUEST, REPLY exchange. That is presented on the following figure.


```

1970.1.1 4:4:25 Client Notice Detected iface eth0.0/3, MAC=0:1d:7e:bc:41:2a.
1970.1.1 4:4:25 Client Notice Detected iface eth0/2, MAC=0:1d:7e:bc:41:2a.
1970.1.1 4:4:25 Client Notice Detected iface lo/1, MAC=0:0:0:0:0:0.
1970.1.1 4:4:25 Client Notice Parsing /etc/dibbler/client.conf config file...
4:25 Client Critical Experimental features are allowed.
4:25 Client Debug Tunnel-mode defined (vendor-id=1727), mapping-prefix enabled too.
4:25 Client Notice Experimental: mappix-prefix enabled
4:25 Client Debug Notify scripts enabled.
4:25 Client Debug Prefix delegation option found.
4:25 Client Debug Parsing /etc/dibbler/client.conf done, result=0
4:25 Client Debug 1 interface(s) specified in /etc/dibbler/client.conf
4:25 Client Info Interface br-lan/5 configuration has been loaded.
4:25 Client Info My DUID is 0:1:0:1:c7:92:cc:55:0:1d:7e:bc:41:2a.
4:25 Client Info Loading old address database (client-AddrMgr.xml), using built-in routines.
4:25 Client Warning Unable to open client-AddrMgr.xml.
4:25 Client Debug Bind reuse enabled (multiple instances allowed).
4:25 Client Notice Creating control (::) socket on the lo/1 interface.
4:25 Client Notice Creating socket (addr=fe80::21d:7eff:febc:412a) on the br-lan/5 interface.
4:25 Client Info Socket bound to fe80::21d:7eff:febc:412a/port=546
4:25 Client Info Creating SOLICIT message with 1 IA(s), no TA and 1 PD(s) on br-lan/5 interface.
4:25 Client Debug Sleeping for 1 second(s).
4:26 Client Info Processing msg (SOLICIT,transID=0xa8414c,opts: 1 3 25 17 8 6)
4:26 Client Debug Sending SOLICIT on br-lan/5 to multicast.
4:26 Client Debug Sleeping for 1 second(s).
4:26 Client Debug Received 283 bytes on interface br-lan/5 (socket=7, addr=fe80::208:54ff:fe04:a324.).
4:26 Client Info Received ADVERTISE on br-lan/5,TransID=0xa8414c, 6 opts: 1 3 25 17 2 7
4:26 Client Debug Sleeping for 1 second(s).
4:27 Client Info Processing msg (SOLICIT,transID=0xa8414c,opts: 1 3 25 17 8 6)
4:27 Client Info Creating REQUEST. Backup server list contains 1 server(s).
4:27 Client Debug Advertise from Server ID=0:1:0:1:e:8d:a2:d7:0:8:54:4:a3:24, preference=0.[using this]
4:28 Client Debug Sleeping for 1 second(s).
4:29 Client Info Processing msg (REQUEST,transID=0x9435c6,opts: 1 3 25 17 8 6 2)
4:29 Client Debug Sending REQUEST on br-lan/5 to multicast.
4:29 Client Debug Sleeping for 1 second(s).
4:29 Client Debug Received 220 bytes on interface br-lan/5 (socket=7,addr=fe80::208:54ff:fe04:a324.).
4:29 Client Info Received REPLY on br-lan/5,TransID=0x9435c6, 5 opts: 1 3 25 17 2
4:29 Client Notice Address 2000::1/64 added to br-lan/5 interface.
4:29 Client Debug RENEW will be sent (T1) after 3600, REBIND (T2) after 5400 seconds.
4:29 Client Debug PD: Adding PD (iaid=1) to addrDB.
4:29 Client Debug PD: Adding 3000::3d7c:0:0 prefix to PD (iaid=1) to addrDB.
4:29 Client Notice Executing external command to ADD prefix: sh ./mappingprefixadd 3000::3d7c:0:0
This is a /var/lib/dibbler/mappingprefixadd script. Is is obsolete. Please use notify script.
4:29 Client Notice ReturnCode = 0
4:29 Client Debug RENEW will be sent (T1) after 3600, REBIND (T2) after 5400 seconds.
4:29 Client Info Tunnel: Tunnel type set to 2
4:29 Client Info Tunnel: Tunnel endpoint is 2000::100
4:29 Client Debug Tunnel: tunnel set to type=2, remote end=2000::100 on the br-lan/5 interface.
4:29 Client Info About to execute command: sh ./notify 2000::1 3000::3d7c:0:0 96 2000::100 2 add
This is /var/lib/dibbler/notify script.
Adding ipip6 tunnel: local endpoint=2000::1, remote endpoint=2000::100
Trying to remove old ipip6 tunnel
Enabling IPv4 forwarding
Enabling IPv6 forwarding
Loading tunnel6 and ip6-tunnel modules
insmod: cannot insert '/lib/modules/2.6.25.16/tunnel6.ko': Success (17)
insmod: Loading module failed: No such file or directory
Creating tunnel ipip6
Bringing up tunnel ipip6
Configuring default route via ipip6
4:29 Client Info Return code=0
4:29 Client Debug Sleeping for 3 second(s).
4:32 Client Debug Sleeping for 1 second(s).
4:33 Client Debug DAD finished successfully. Address 2000::1 is not tentative.
4:33 Client Debug Sleeping for 3596 second(s).

```

9.3 Remote ip46nat configuration validation

After changing tunnel-type (from 2 to 1) in the server.conf on PC2, last test was repeated. After receiving requested parameters from the server, external scripts were called. This time dibbler-client printed slightly different messages. Log from this execution is presented below.

```

root@OpenWrt:/tmp/lib/dibbler# dibbler-client run
| Dibbler - a portable DHCPv6, version 0.7.2 (CLIENT, Linux port)
| Authors : Tomasz Mrugalski<thomson(at)klub.com.pl>,Marek Senderski<msend(at)o2.pl>
| Licence : GNU GPL v2 only. Developed at Gdansk University of Technology.
| Homepage: http://klub.com.pl/dhcpv6/
1970.1.1 4:2:18 Client Notice My pid (1175) is stored in /var/lib/dibbler/client.pid
1970.1.1 4:2:18 Client Notice Detected iface ip6tnl0/6, MAC=0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0.
1970.1.1 4:2:18 Client Notice Detected iface br-lan/5, MAC=0:1d:7e:bc:41:2a.
1970.1.1 4:2:18 Client Notice Detected iface eth0.1/4, MAC=0:1d:7e:bc:41:2a.
1970.1.1 4:2:18 Client Notice Detected iface eth0.0/3, MAC=0:1d:7e:bc:41:2a.
1970.1.1 4:2:18 Client Notice Detected iface eth0/2, MAC=0:1d:7e:bc:41:2a.
1970.1.1 4:2:18 Client Notice Detected iface lo/1, MAC=0:0:0:0:0:0.

```

```

1970.1.1 4:2:18 Client Notice Parsing /etc/dibbler/client.conf config file...
2:18 Client Critical Experimental features are allowed.
2:18 Client Debug Tunnel-mode defined (vendor-id=1727), mapping-prefix enabled too.
2:18 Client Notice Experimental: mappix-prefix enabled
2:18 Client Debug Notify scripts enabled.
2:18 Client Debug Prefix delegation option found.
2:18 Client Debug Parsing /etc/dibbler/client.conf done, result=0
2:18 Client Debug 1 interface(s) specified in /etc/dibbler/client.conf
2:18 Client Info Interface br-lan/5 configuration has been loaded.
2:18 Client Info My DUID is 0:1:0:1:c7:92:cc:55:0:1d:7e:bc:41:2a.
2:18 Client Info Loading old address database (client-AddrMgr.xml), using built-in routines.
2:18 Client Warning Unable to open client-AddrMgr.xml.
2:18 Client Debug Bind reuse enabled (multiple instances allowed).
2:18 Client Notice Creating control (::) socket on the lo/1 interface.
2:18 Client Notice Creating socket (addr=fe80::21d:7eff:febc:412a) on the br-lan/5 interface.
2:18 Client Info Socket bound to fe80::21d:7eff:febc:412a/port=546
2:18 Client Info Creating SOLICIT message with 1 IA(s), no TA and 1 PD(s) on br-lan/5 interface.
2:18 Client Debug Sleeping for 1 second(s).
2:19 Client Info Processing msg (SOLICIT,transID=0x41268c,opts: 1 3 25 17 8 6)
2:19 Client Debug Sending SOLICIT on br-lan/5 to multicast.
2:19 Client Debug Sleeping for 1 second(s).
2:19 Client Debug Received 283 bytes on interface br-lan/5 (socket=7, addr=fe80::208:54ff:fe04:a324.).
2:19 Client Info Received ADVERTISE on br-lan/5,TransID=0x41268c, 6 opts: 1 3 25 17 2 7
2:19 Client Debug Sleeping for 1 second(s).
2:20 Client Info Processing msg (SOLICIT,transID=0x41268c,opts: 1 3 25 17 8 6)
2:20 Client Info Creating REQUEST. Backup server list contains 1 server(s).
2:20 Client Debug Advertise from Server ID=0:1:0:1:e:8d:a2:d7:0:8:54:4:a3:24, preference=0.[using this]
2:20 Client Debug Sleeping for 1 second(s).
2:21 Client Info Processing msg (REQUEST,transID=0x2b22c5,opts: 1 3 25 17 8 6 2)
2:21 Client Debug Sending REQUEST on br-lan/5 to multicast.
2:21 Client Debug Sleeping for 1 second(s).
2:21 Client Debug Received 220 bytes on interface br-lan/5 (socket=7, addr=fe80::208:54ff:fe04:a324.).
2:21 Client Info Received REPLY on br-lan/5,TransID=0x2b22c5, 5 opts: 1 3 25 17 2
2:21 Client Notice Address 2000::1/64 added to br-lan/5 interface.
2:21 Client Debug RENEW will be sent (T1) after 3600, REBIND (T2) after 5400 seconds.
2:21 Client Debug PD: Adding PD (iaid=1) to addrDB.
2:21 Client Debug PD: Adding 3000::3f6c:0:0 prefix to PD (iaid=1) to addrDB.
2:21 Client Notice Executing external command to ADD prefix: sh ./mappingprefixadd 3000::3f6c:0:0
This is a /var/lib/dibbler/mappingprefixadd script. Is is obsolete. Please use notify script.
2:21 Client Notice ReturnCode = 0
2:21 Client Debug RENEW will be sent (T1) after 3600, REBIND (T2) after 5400 seconds.
2:21 Client Info Tunnel type set to 1
2:21 Client Info Tunnel: Tunnel endpoint is 2000::100
2:21 Client Debug Tunnel: tunnel set to type=1, remote end=2000::100 on the br-lan/5 interface.
2:21 Client Info About to execute command: sh ./notify 2000::1 3000::3f6c:0:0 96 2000::100 1 add
This is /var/lib/dibbler/notify script.
Adding ip46nat tunnel: IPv4:src=10.10.10.0,dst=* -> IPv6:src=3000::3f6c:0:0,dst=2000::100
Removing old ip46nat module
Enabling IPv6 forwarding
Enabling IPv4 forwarding
Inserting ip46nat module
Configuring IPv6 routing
Adding iptables rule
--Kernel output start--
IPv6 dropped (no route): 160
IPv4 dropped (transmission failed): 0
IPv6 dropped (transmission failed): 0
-----
IPv4-IPv6 NAT module unloaded.
IPv4-IPv6 NAT module loaded: v6prefix: 2000::100, v6prefixm: 3000::3f6c:0:0, v4addr: 10.10.10.0
Handlers for IPv4 and IPv6 installed.
#IPv4 rcvd (rcvd so far: 1) [src=192.168.1.100, dst=192.168.1.1,looking for 10.10.10.0/24]
#IPv4 rcvd (rcvd so far: 2) [src=192.168.1.100, dst=192.168.1.1,looking for 10.10.10.0/24]
#IPv4 rcvd (rcvd so far: 3) [src=192.168.1.100, dst=192.168.1.1,looking for 10.10.10.0/24]
--Kernel output end--
2:21 Client Info Return code=0
2:21 Client Debug Sleeping for 3 second(s).
2:24 Client Debug Sleeping for 1 second(s).
2:25 Client Debug DAD finished successfully. Address 2000::1 is not tentative.
2:25 Client Debug Sleeping for 3596 second(s).

```

Conclusion

Due to time constrains, scope and diversity of tests is somewhat limited. However, all use cases are validated. No anomalies were encountered. The code is ready for more extensive testing and field trials.

10 Source code

This section discusses internal architecture for developed or extended software.

10.1 Code overview for ip46nat module

ip46nat module was developed from scratch. Since there is one specific use of its operation, all module parameters are specified during module loading. Module loading operation is handled by the `hello_init()` function. It checks if all required parameters are provided and well formed. After the check is successful, it calls `register_handlers()` function. Most packet handling in the kernel code is done via handlers. Once packet is received from the network, all handlers for specific packet type are called. In this case, there are 2 handlers defined and used: `ipv4_handler` and `ipv6_handler`. Both functions check if the received traffic meet expected criteria, i.e. their address parameters are as expected. Once the packet is checked and is considered to perform NAT, one of 2 separate functions is called: `ipv4_send_as_ipv6` or `ipv6_send_as_ipv4`. In both cases packet header is rewritten, TCP/UDP checksum recalculated and routing for a new packets is selected. If the routing is present, packet is finally sent. Missing routing may means that:

- there is no routing configured for this destination address
- Packets using 0.0.x.x class as a source address, will not be sent.
- Traffic forwarding is not enabled

11 Best practices and debugging tips

This section provides recommendations, useful observations and best practices.

- Before you start any risky firmware upgrade, make sure that `boot_wait` is enabled (see section 3.5). Firmware with Linux kernel 2.4 is required.
- Network configuration is specified in the `/etc/config/network` file. It contains information regarding port assignments, used IPv4 addresses and vlan assignments. For details, see section 4.
- When modifying scripts for OpenWRT, keep in mind that it uses `ash` instead of `bash`. The differences are rather minimal (e.g. different function definitions). For details regarding `ash` environment, see: <http://www.thelinuxblog.com/linux-man-pages/1/ash>

12 Links

Following links are recommended reading:

- <http://klub.com.pl/ip46nat/> - ip46nat project homepage
- <http://openwrt.org/> - OpenWRT project website
- <http://downloads.openwrt.org/kamikaze/docs/openwrt.html> - OpenWRT manual
- <http://klub.com.pl/dhcpv6/> - Dibbler homepage. User's Guide, Developer's Guide and a dibbler source code is available here.
- <http://www.thelinuxblog.com/linux-man-pages/1/ash> – `ash` (bash-like shell replacement) manual page
- <http://www.linuxfoundation.org/en/Net:Iproute2> – `iproute` homepage

- <http://devresources.linux-foundation.org/dev/iproute2/download/> - iproute source code download

13 Contact

Author of this project can be reached using e-mail <mailto:tomasz.mrugalski@gmail.com>.