Acta Cybernetica **15** (2002) 683–704.

Development of a Communication Environment between IPv6 and IPv4

Gábor Fóris*, László Sógor†, Péter Hendlein‡, Krisztián Notaisz§ and Márta Fidrich¶

Abstract

The aim of this paper is to present the design, specification, implementation and testing of a demonstration environment for examining a genuinely new communication technique. This technique ensures that 3G mobile networks can communicate with legacy Internet phones. More than one levels of the TCP/IP protocol family are necessary for the communication, so we had to develop device drivers and user level applications too. The different levels require various development techniques and tools, whose efficiently combined usage is emphasized.

Keywords: SIP, SDP, MEGACO, IPv6, NAPT-PT

1 Motivation

The Internet Protocol version 4 (IPv4) [1] has been available since 1981. As it can be seen today, its success is indisputable. However, the rapid growth of the Internet has created a number of problems for the administration and operation of the global network, such as the quite limited address space, which will be exhausted in the near future. The new version 6 of the Internet Protocol (IPv6) [2] includes expanded addressing capabilities, header format simplification, improved support for extensions and options, plug and play services, authentication and privacy capabilities, native mobility support, and also real-time and quality services [3]. As the world continuously moves to IPv6, it is essential to solve the communication problems between the two network-layer protocols.

*GE Medical Systems, Lajos u. 48-66, H-1036 Budapest Hungary, e-mail: gabor.foris@med.ge.com

†GE Medical Systems, Lajos u. 48-66, H-1036 Budapest Hungary, e-mail: laszlo.sogor@med.ge.com

‡Axelero Tiszanet, Fekete Sas u. 28, H-6720 Szeged Hungary, e-mail: hendlein@tiszanet.hu

§GE Medical Systems, Lajos u. 48-66, H-1036 Budapest Hungary, e-mail: krisztian.notaisz@med.ge.com

¶Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanuk tere 1, H-6720 Szeged Hungary, e-mail: fidrich@sol.cc.u-szeged.hu

Since IPv6 was chosen to be the network protocol for Third Generation (3G) Mobile Networks, an urgent demand appeared in connecting the existing IPv4-based networks to the new one. Although IPv6 was designed to extend IPv4, the extension incorporates so many fundamental changes having far-reaching impacts, that IPv6 is not compatible with IPv4. It means that the IPv4-based programs need to be modified and recompiled to support IPv6 and there is no way of direct communication between IPv4-only and IPv6-only systems.

Because of the large installed base of IPv4 hosts and routers, the changeover of protocols will take still some years. During the long process of transition – which is agreed to be not easy – it is needed that programs in both realms can communicate with each other. IP-level gateways only are not enough for successful communication, since IP addresses may also be important information carried in application layer protocols, as in FTP, SIP and SDP. That is, the change of the two protocols inherently affects applications, so different level gateways (network, transport and application) have to be developed between them.

2 Networking description of the task

The 3G networks use SIP [4] (Session Initiation Protocol) as their call control protocol and IPv6 as network layer protocol for wireless communication. Currently IPv4 is used as network protocol for the Internet. There are two ways to ensure the communication (see [15]) : SIP-ALG (SIP-Application Level Gateway) and IP-level gateway remotely controlled by a SIP proxy. The remotely controlled IP-level gateway is a better solution, because rewriting the IP translator is not needed when the SIP protocol changes, it is sufficient to modify only the SIP implementations.

The goal of our work is to create a demonstration system, which connects a mobile SIP User Agent based on IPv6 and another SIP User Agent based on IPv4 via two SIP proxies (IPv6 and IPv4) and NAPT-PT [5] (Network Address Port Translator - Protocol Translator). Indeed, the real SIP communication between the IPv4 and the IPv6 networks means the communication between the two proxies via the NAPT-PT. When a media connection is created by the SIP server and client (with the help of SIP proxies), the IPv6 SIP proxy (Media Gateway Controller) sends a command using the MEGACO [7] (MEdia GAteway COntrol) protocol to add a binding to the NAPT-PT (Media Gateway). This binding helps the NAPT-PT to convert the packets transporting the media connection. When the media connection is terminated, the IPv6 SIP proxy sends a message to the NAPT-PT to release the binding. The NAPT-PT also has a DNS-ALG [8] (DNS-Application Level Gateway) extension to convert DNS queries and responses between the IPv4 and IPv6 networks.

3 Concepts of development

Due to its free source code and available documentation, the Linux operating system was chosen as a development and test platform. We worked on five computers

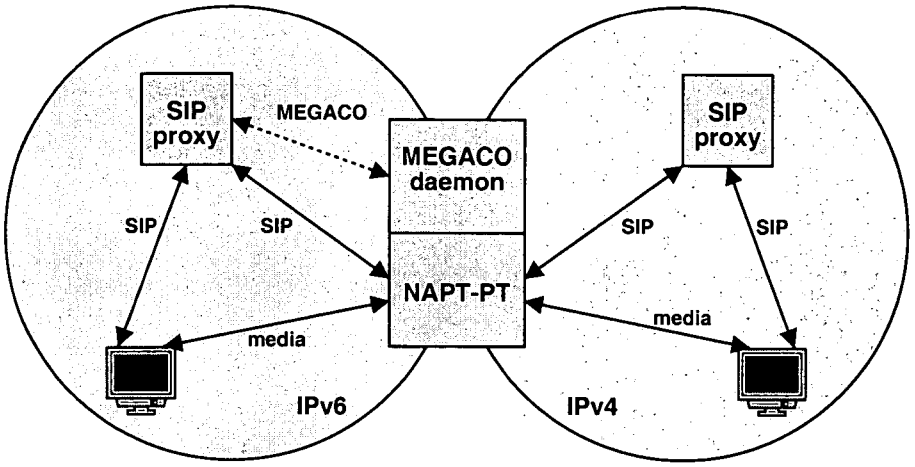

Figure 1: Test environment for multimedia call initiation between IPv4 and IPv6

having Intel Celeron processor at speed of 466 MHz using 128 MB of SDRAM at speed of 66 MHz. Each computer had two 3com 905 10/100Mb/s Ethernet cards, providing an IPv4 and an IPv4 / IPv6 interface. (Linux has dual stack IPv6 implementation, that is why we mentioned the latter interface to be dual, although we used it as a "pure" IPv6 interface.) All the PCs run GNU / Debian Linux; we relied on the last stable kernel version 2.2.17. We used the C/C++ programming languages (C for kernel level development and C++ for others).

The communication environment introduced in Section 2 requires the development of three entities, thus our work was divided into three parts: the SIP-specific NAPT-PT with DNS-ALG, the SIP proxies, and the NAPT-PT controlling MEGACO daemon were developed parallelly. Please see the table below for a summary.

part	program. level and networking layers	connections
NAPT-PT with DNS-ALG	kernel level in C network, transport and application layer	MEGACO daemon (IOCTL calls) SIP proxies (SIP commands) SIP user agents (media streams)
IPv4 SIP proxy	user level in C++ application layer	NAPT-PT (SIP commands) IPv4 SIP user agents (media & SIP)
IPv6 SIP proxy	user level in C++ application layer	NAPT-PT (SIP commands) IPv6 SIP user agents (media & SIP) MEGACO daemon (MEGACO prot.)
MEGACO daemon	user level in C++ application layer	NAPT-PT (IOCTL calls) IPv6 SIP proxy (MEGACO protocol)

1. Media Gateway
   It is a special router that knows both IPv4 and IPv6 protocols. It converts the IPv6-based TCP / UDP packages into IPv4-based packages (exchanging address and port) with the help of its inner table, and also handles SIP and DNS queries passing through. Since NAPT-PT is an IP-level packet translation mechanism and it requires kernel-level programming, we had to develop it in C programming language.

2. Media Gateway Controller
   Media connections are initiated by SIP, and it is the IPv6-based SIP proxy that controls package conversion of the NAPT-PT. SIP proxies are application-level daemons. These daemons were developed in C++ because we could use some ready-made object libraries developed for SIP over IPv4.

3. Way of Controlling
   MEGACO protocol ensures the controlling commands between NAPT-PT and IPv6-based SIP proxy at two different levels: MEGACO library aiming language recognition is used directly by the SIP proxy, while its application, the MEGACO daemon controls NAPT-PT with the help of IOCTL calls. MEGACO is a heavy protocol with highly complex grammar which we transformed into LL(k) grammar to make it consumable for the ANTLR parser generator. The object oriented programming technique is very suitable for modeling this grammar. We chose the C++ programming language because controlling of NAPT-PT requires IOCTL calls, which are available only in C/C++.

As it can be seen by now, our communication environment is a fairly complex system composed of three related yet distinct entities. Our software engineering task was not usual: we had to develop parts from kernel to user level, from network to application layer of the TCP / IP protocol family, in C and C++ languages. Indeed, communication of our software pieces required novel solutions and expert techniques. We could make good use of various development tools, whose roles are emphasized. The contribution of our work is not only the summary of a genuinely new technique to ensure communication between 3G mobile networks and legacy Internet phones. It also comes from the presentation of various software development concepts based on our consensus.

Although Internet phone programs [18] are available presently, they are based on IPv4. As far as we know, there is no software for IPv6-based phone. Also, we do not know about any implemented system (either already working or under research), which connects 3G mobile and Internet phones.

The rest of this article is organized as follows. We briefly describe commonly used tools in Section 4. After that we present the three development parts: the NAPT-PT, MEGACO and SIP. Finally, we give a conclusion, also future research is highlighted.

4 Commonly used tools

All the three groups used CVS [9] for managing the parallelly developed code. CVS is the Concurrent Versions System, the dominant open-source network-transparent version control system. CVS is useful for everyone from individual developers to large distributed teams:

- Its client-server access method lets developers access the latest code from anywhere using Internet connection.
- The management of conflicts, which can be caused by the unreserved check-out model to version control, is supported.
- Its client tools are available on most platforms.

We made good use of UML, the Unified Modeling Language at the planning of the classes. For the UML modeling we have chosen Together [10], a JAVA based development program, because it is available on various platforms, including Linux. The core features of Together are:

- Simultaneous round-trip engineering: Java, EJBs, IDL, and C++
- Multi-level, flexible documentation generation
- Multi-user team support across the enterprise
- Large project support

We needed a network traffic analyzer program for testing the conversion algorithms (NAPT-PT and DNS-ALG) and monitoring the network traffic. We chose Ethereal [16], a "sniffer" available on Unix-like operating systems. It uses GTK+, a graphical user interface library, and libpcap, a packet capture and filtering library. Ethereal knows both IPv4 and IPv6 network-layer protocols, TCP and UDP transport-layer protocols and some application-layer protocols, for example DNS and HTTP. For an Ethereal snapshot, see Fig 2.

5 NAPT-PT

5.1 Implementation form of NAPT-PT

Unlike a typical monolithic kernel, the Linux kernel has a feature to be able to load and run kernel extensions (for example device drivers) without re-linking the kernel and restarting the system. These extensions are called modules. Loading and removing of modules can be controlled manually (the system administrator may load and remove modules using `insmod`, `rmmod` and `modprobe` programs) and automatically (when the system needs it, it is loaded).

As NAPT-PT (Network Address Port Translator - Protocol Translator) is a special IP packet conversion algorithm that needs services of the kernel (functions, data structures, etc.), it has been implemented at kernel-level. We chose the module form because it is not necessary to reboot the system when we want to run our new code, it is enough to remove the old module and load the new one.

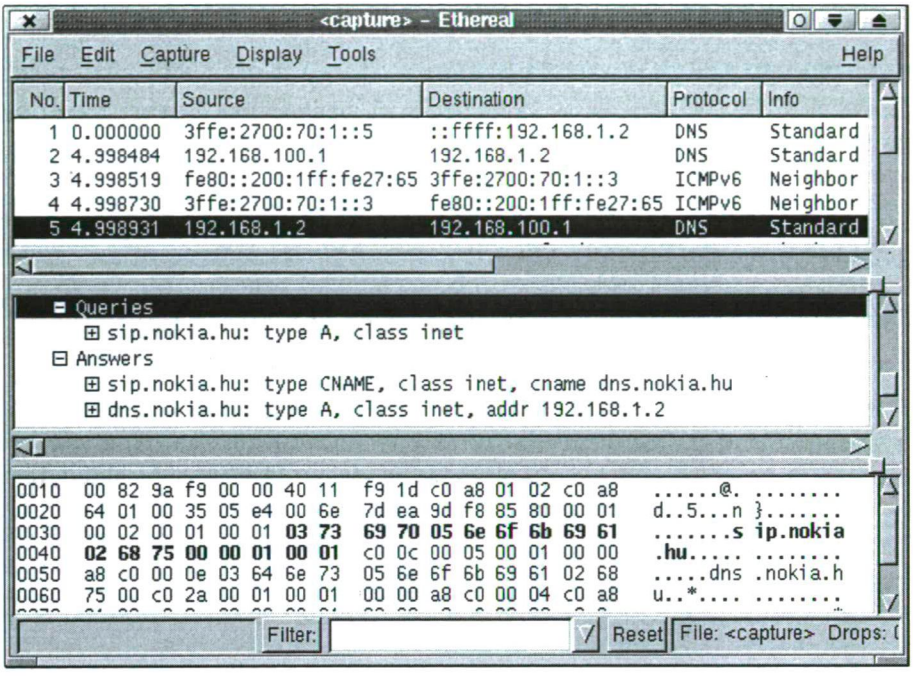

Figure 2: The Ethereal network traffic analyzer

5.2 NAPT-PT as network device

We implemented NAPT-PT as a pseudo network device, because in this way packets that needs conversion can be simply routed to it using the standard `route` procedure. Moreover, network devices can be easily configured with the `ifconfig` utility. All utilities used to manipulate pseudo network devices are part of standard Linux distributions; direct interaction between a user-level application and the loaded module is also possible with the `ioctl` system call. More details about networking in the Linux kernel can be found in [22].

5.3 Parts of NAPT-PT

Our NAPT-PT implementation consists of the following parts: packet conversion, conversion database, configuration interface and DNS-ALG.

- Packet conversion: every IPv4 and IPv6 packets that need to be converted go through this part. First this code determines the type of the packet: DNS, SIP or media stream. If the packet is a DNS one, the conversion procedure will pass it to the DNS-ALG module which translates is. The handling of a SIP packet is trivial, it does not require special algorithm just forwarding.

So as to convert a media stream packet (IPv4 packets to IPv6, and vica versa), data are needed from the conversion database. The actual IPv4 – IPv6 conversion procedure is described in our previous article [17].

- The conversion database contains records with the following fields: IPv6 address / port, IPv4 address / port and protocol id. It is important to find the record having the requested properties quickly, so we had to find a data structure which could be easily implemented at kernel level and was fast enough. The static hash of chained lists has been chosen.

- Configuration interface: configuration data (addresses of IPv6 DNS server and SIP proxy as well IPv4 address pool of NAPT-PT) must be given to NAPT-PT for the proper operation. In addition to this, IPv6 address / port / protocol triplets have to be given to NAPT-PT as a binding requests and the NAPT-PT has to response with IPv4 address / port pairs. The configuration and binding data must be given by user level processes (configuration utility and MEGACO daemon). On UNIX systems, the user-level programs can control kernel-level drivers via the IOCTL interface. The user-level process has to send the kernel a triplet mentioned before, which takes 19 bytes (16 + 2 + 1). The type of IOCTL, which is to be used for network devices, can carry 16 bytes of data, so we have to use a pointer. Another difficulty is that the kernel- and user-level memories are separated: a user-level memory cell cannot be reached form kernel-level directly and vica versa. There is only one possibility to copy data between kernel- and user-level: usage of `copy_from_user` and `copy_to_user` functions.

- DNS-ALG: this extension converts the DNS queries and responses between IPv4 and IPv6 using the configuration data of NAPT-PT. Its base functionality is to convert the DNS payload, which contains the following parts: header, questions and resource records; see [8]. The IPv4 addresses are stored in A resource records, while the IPv6 ones are stored in AAAA resource records. We assume that the length of the DNS payload is less than or equal to 512 bytes. (because of using only UDP)

  Only the IPv6 DNS server and only the IPv6 SIP proxy can be reached from the IPv4-side of the NAPT-PT using the first usable address of the IPv4 address pool of the NAPT-PT. So in the case of IPv6 $\rightarrow$ IPv4 and AAAA $\rightarrow$ A RR-conversion, the RDATA of the A RR will be first_addr (32 bit, network byte order).

  We note that source DNS question / Resource Record / domain / header means the DNS question / RR / domain / header that we convert. Destination DNS question / RR / domain / header means the result of the conversion.

  When NAPT-PT recognizes a DNS packet, it passes the DNS payload to DNS-ALG module that does the conversion. The DNS payload is a rather complex structure with lots of variable length parts.

  The payload of a DNS packet contains the following parts:

1. header, this contains some fields and numbers of queries and Resource Records (RR)
2. queries
3. answer RRs
4. authority RRs
5. additional RRs

The DNS-ALG has to convert the header, the questions and RRs.

During the planning and implementation of NAPT-PT we created the entry points of DNS-ALG, so we had to design only the conversion procedure itself. There are two main conversion functions, one for IPv6 → IPv4 and another one for IPv4 → IPv6 direction: *convert_dns_64* and *convert_dns_46.*

5.4 Restrictions

- The NAPT-PT uses an IPv4 address pool and distribute ports from this to the media connections. This pool will be set by an IPv4 network address/netmask pair.
- The network and broadcast addresses will not be distributed by NAPT-PT.
- The IPv6 DNS server and SIP proxy can be reached from IPv4 network using the first IPv4 address of the NAPT-PT's address pool, that is:
  -if its netmask is 255.255.255.255, then this IPv4 address is supposed to be used for DNS and SIP queries
  -if the netmask is 255.255.255.254(2 addresses), then we return a configurational error, because we did not get an effective address
  -otherwise, we use the first effective address (the one after the network address) for DNS/SIP queries
- The distribution goes as follows:
  -ports between 6000-65531 will be handed out for TCP and UDP connections alike
  -first the ports between 6000-65535 of the first effective address will be handed out, the the next, ... and after the last one we use the first one again.
- The handling of media packages is special:
  -in case of UDP, the first effective address/port 5999 will be the source address/port in the 6 → 4 direction; in the other direction there is no such problem.
- There is no restriction upon the IP addresses
  We do not use a table for storing the free ports. If needed, we look up a free port in the NAPT-PT table.
- In case of RTP protocol, we need a double entry and for this we will always use even ports together with the next odd one.

6 MEGACO daemon

6.1 Base problem

MEGACO protocol (MEdia GAteway COntrol) is used to control a media gateway (MG) by a media gateway controller (MGC). In our project, the NAPT-PT is a MG (converts packets of the media from IPv4 to IPv6 and vica versa) and the SIP proxy is a MGC (sends requests to NAPT-PT to reserve or free IPv4 address/port in MG). Indeed, the SIP proxy uses MEGACO protocol to control the NAPT-PT, or in other words, the communication of NAPT-PT and SIP proxy is resolved by MEGACO.

We decided to separate the language recognition and the controlling parts, because the language recognition functionality is the same in the NAPT-PT and SIP proxy as well. To do so, we first created a C/C++ library, whose basic task is the recognition of the language. It handles MEGACO messages and also provides a flexible data structure (like C++ class hierarchy). Control of the MG by a MGC is achieved with the help of this MEGACO library. The library can be used in two ways depending whether the NAPT-PT or the SIP proxy wants to use it. NAPT-PT works at the network and transport layer, so it needs a daemon at the application layer, which is the layer of the MEGACO-based communication. Thus we had to develop a MEGACO daemon as well. This daemon handles MEGACO requests & answers and controls NAPT-PT by IOCTL calls. SIP proxy is capable of application-layer communication, thus it does not need any further utility. It makes use the MEGACO library directly in its media controlling part.

6.2 Design of MEGACO library

The MEGACO library is composed of three parts:

1. The core contains a manageable data structure instead of messages having difficult grammar.

2. The message parser analyzes an arrived message and creates data structure from it.

3. The data transformer generates message from the data structure.

The most important feature of the core is to ensure the manageable data structure (C++ class hierarchy) with handler functions. Each class matches to one item of the MEGACO grammar in such a way that the hierarchy of grammar elements are modeled. These classes are needed to be filled up with given values (during message parsing) and this is the task of the handler functions. Handler functions are also used to generate messages from the data structure.

The classes are grouped in some packages like

- MegacoMain: this is the base class
- Messages: classes, which realize main MEGACO messages
- Transactions: classes, which implement MEGACO transactions
- Actions: classes implementing MEGACO actions
- Commands: classes, which implement MEGACO commands
- Descriptors: classes implementing MEGACO descriptors
- SDP: this contains the simplified SDP [6] data, because in MEGACO some grammar elements may include SDP payload.

Now let us see how to generate MEGACO message from a data structure. We obtain the message string by calling the generator function of the MegacoMessage class. The final result is a string, which will be sent in the message. Traversing the attributes of the data structure makes the generation easy: concat (in this order) the start text of an object, the attributes and the end text. If an attribute is an object, call its generator function to get the value of the object. If the attribute of an object is not a further object, we put its own specific text (for example "id=value") instead of calling the generator function of attribute.

In the message analysing part we have to decide about each message whether it is an implemented MEGACO message or not. That is, the Media Gateway (MG) – or on the other side: the Media Gateway Controller (MGC) – is able to process it (i.e. to create data structure from it) or not. Since it is the most complex (and that is why the longest) part of the library, we describe its development separately in the next subsections.

6.2.1 Parser

The base problem, we had to solve, was creating a message analyzer, which has to process all incoming messages and handle the commands these contain. Our solution relies on the core of the library, which implements an object structure by covering the graph of all recognizable messages. The analyzer itself is called by the MEGACO daemon (see later), and it has to fill the object structure that represents the construction of the received message. We decided to use a parser generator to make the system more flexible and easy-to-modify.

Every code generator needs an input grammar, which mostly published in some kind of BNF form. The Backus-Naur Form (BNF) is a convenient means for writing down the grammar of a context-free language. Augmented Backus Naur Form (ABNF) [23] differs from BNF in naming rules, repetition, alternatives, order-independence, and value-ranges. The Extended Backus-Naur Form (EBNF) [24] adds the regular expression syntax of regular languages to the BNF notation, in order to allow very compact specifications.

There are lots of parser generators (YACC, ANTLR, etc.), which differ in the expectation of the type of the input grammar and the programming language of the generated code. We chose ANTLR [14] version 1.33 – also known as PCCTS – to recognize MEGACO messages because this tool can generate C++ source code, and it expects an LL(k) grammar, given in EBNF syntax, as input.[1] The generated

[1]We note that YACC expects an LALR(1) grammar: it generates a quite fast parser; however,

source code is capable to decide whether its input is an element of the language or not — this feature is indeed used for examining the syntactical correctness of the input text.

In the input file of ANTLR we have to define tokens — for the lexical analyzer — and we have to give rules - for the syntactic analyzer. To implement a code analyzer, first a lexical analyzer is used to filter the input data and to eliminate those parts of it that will not be used further on. The valuable parts can be packed into tokens, and passed on to the syntactic analyzer, which is the parser itself. The parser contains the rules of the input grammar. These rules are identified by non-terminals, and the analyzing process consists of fitting these rules. From these non-terminals ANTLR generates functions, which can receive parameters by value or by address. There is an ability in this tool to specify actions for each recognized part of the grammar. This is useful, because we can put code into the parser directly, by invoking formerly implemented functions or adding own code written in the target programming language. (The latter one generates the proper data structure about each received message.)

6.2.2 Problems and solutions

The first task we had to solve is to convert the MEGACO grammar from ABNF (in which it is published) into EBNF required by ANTLR. ABNF has such rules that could not be easily transformed into EBNF form (eg. when a non-terminal's number of repetitions is limited). These rules were extended and further checking were implemented in the source code. In this process we also had to face how annoying to replace all the arguments signed in ABNF form with its real sequence. After finishing this part, compiling by ANTLR resulted in tremendous ambiguous rule errors. This problem occurred because the input grammar was not LL(k).

To overcome this problem, we constructed new rules and transformed the already stated rules and tokens as well to get an LL(k) grammar.

6.2.3 Resolving SDP payload

The message analyzer is composed of two parts. The MEGACO parser, which recognizes all MEGACO valid messages, and a smaller one, the SDP parser, which is invoked from the MEGACO parser when needed. SDP stands for Session Description Protocol, its roles are describing properties of multimedia sessions, for example protocol, port, origin and description. Although the grammar of the SDP protocol is much smaller and less complicated than the one of MEGACO, we had to follow the same way an "LL(k)-ization" process for SDP grammar as for MEGACO grammar.

LALR(1) means some restrictions on the input grammar (or expects transferring work from the programmer).

6.3 Design of MEGACO daemon

First, we extended the MEGACO library with network managing interface, which sends and receives MEGACO messages. According to the RFC, MEGACO has to ensure message transfer, that is: in case of successful reception, it has to acknowledge it. In case of (possible) loss, it has to resend the message and it also has to drop the repetitions. We included this functionality in the network interface, which is needed for the SIP proxy and for the MEGACO daemon (controlling NAPT-PT). This interface calls the parser in the MEGACO library and if the analysed message is wrong, that is, not an element of the language generated by the MEGACO grammar, it replies an error message. If the analysed message is accepted, the interface sends an acknowledgment.

Now, let us suppose that the incoming message is accepted, i.e. element of the language. In this case the network interface passes the message to the daemon. It interprets and processes the message: e.g. it registers a new binding in the NAPT-PT or resolves a binding. If the requested action was successful, the daemon stores / removes information on binding and makes a reply message for the network interface. Otherwise, the daemon makes an error message for the network interface.

6.4 Functionality of MEGACO daemon

We had to separate the MEGACO daemon into several independent parts. The first part is the "Listener". Listener is waiting for incoming TCP sockets, and handles them. The second part is the MEGACO library extension, the MEGACO communication unit, we call it the "'Handler". It works with an incoming socket, reads the package from it, sends an acknowledgment to it, and forwards the MEGACO data to the Processor unit. The "Processor" analyzes this data and registers the resources in NAPT-PT. The result of registration is sent back to "Handler", and data is saved to the "Central data" unit. The core part is the "Daemon". It initializes NAPT-PT and "Listener", sends registration to IPv6 SIP proxy (using a "Handler").

Figure 3 presents the above introduced units and operations between them. Dotted-line rectangles are modeling one thread in the daemon. Here is the description of communication signaled by numbered operational arrows:

1. IOCTL
2. Initialization
3. Request to MGC (eg. Service-Change command)
4. Reply from MGC (eg. reply to Service-Change command)
5. Accepting connection, creates thread (parameter: the socket)
6. Request from MGC (eg. Add command)
7. Reply to MGC (eg. reply to Add command)
8. IOCTL
9. Reads from and writes to central data

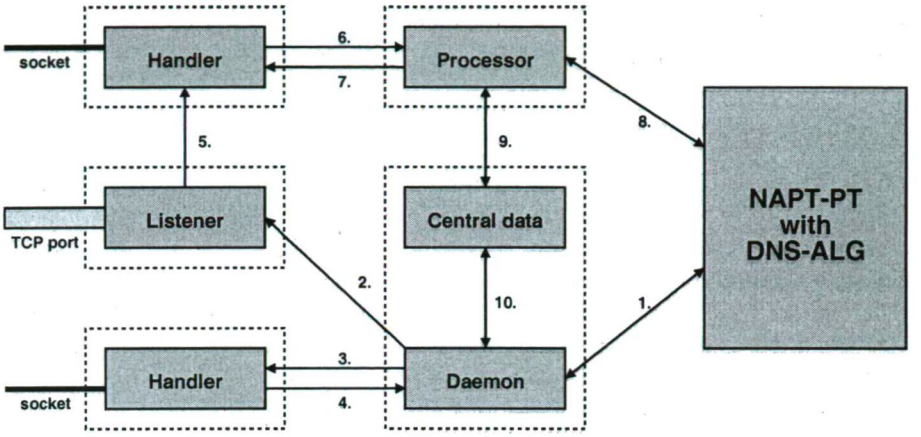

Figure 3: Main operations

10. Reads from and writes to central data

6.5 Main Restrictions

- The connection between MG (Media Gateway = NAPT-PT) and MGC (Media Gateway Controller = IPv6 SIP proxy) is based on either TCP or UDP. We have chosen TCP for the simplification of the implementation.
- We implement the text-encoding version of MEGACO.
- We formalize the grammar with PCCTS grammar analyzer, thus the not implemented commands can be inserted easily.
- We create a library, which can be used both at the implementation of MEGACO daemon and at the implementation of SIP proxy.

7 SIP proxy

The SIP protocol is used to control media sessions described by SDP [6]. We had to develop two SIP proxies, an IPv4 and an IPv6 one, the latter with MEGACO support to control the media gateway (NAPT-PT). The media connections are initiated by SIP, and the IPv6-based SIP proxy controls the NAPT-PT package conversion through adding / releasing bindings for media sessions using the MEGACO protocol.

To build the two proxies we deployed some freely available programs and libraries to reduce the development time. For example, we modified and extended

the Dissipate library – a basic SIP implementation – and Kphone [18], a SIP telephone program built on the top of Dissipate, to fit for our purposes.

7.1 Qt

Both the Dissipate library and the Kphone use the Qt library [19]. QT is a cross-platform C++ GUI framework. It provides a number of classes, for example data structures (queue, stack, list), advanced string handler functions, pattern matching functions and functions to make the building of a GUI easier.

This library indeed provided us a lot of useful classes. For example, we could make very good use of the QString class, with plenty of pattern matching functions. The parsing of the configuration file of the proxy is built on these functions. We also used some higher level data structure, like QList and QFile.

7.2 Dissipate and Kphone

Since the size of the two libraries are huge, we had to find a way to make the browsing and the understanding of the operation of the source code easy. This problem was solved by Doxygen [20]. Doxygen is a documentation system for C++, Java, IDL and C[2]. It can help you in several ways but the most useful feature of the program is that it can generate an on-line documentation (in HTML) and / or an off-line reference manual (e.g. in PDF, LaTeX) from a set of documented source files. The key to make this program really valuable is that the programmer has to put very telling comments in the source!

We also had to port the Kphone program and the Dissipate library to IPv6. This process went like this:

- Making a _v6 tagged duplicate of the original directories:
  Renaming the files and rewriting the references in the files. Browsing and rewriting the Makefiles, to enable proper compilation and installation of the Dissipate_v6 library and Kphone_v6. After this, the Kphone used the Dissipate library, the Kphone_v6 used the Dissipate_v6 library, although the code in the two libraries were yet the same.

- Locating the parts relevant to IPv6 sockets and appropriating it to IPv6 standards.

- For the cooperation with NAPT-PT and for the compatibility with RTP/RTCP [21], we had to reserve two ports for the media. However, Kphone, originally, reserves only one port.

- Correcting a bug in the Dissipate source: in extreme cases, the original code does not releases certain ports. For further details see [25].

[2]Doxygen is developed under Linux, but is set-up to be highly portable. As a result, it runs on most other UNIX flavors as well. Furthermore, an executable for Windows 9x/NT is also available.

7.3 IPv4 SIP proxy

Thinking in advance, we decided to build the IPv4 SIP proxy in a rational way to ease our work for creating the IPv6 SIP proxy. This meant that we developed a basic stateless SIP proxy to handle SIP requests and responses. Care was taken to structure the proxy in small reusable functions that might occur in other parts of the source or even in the IPv6 proxy. The proxy also makes use of the dissipate library, just as Kphone does.
We designed several classes to store and handle SIP registrations. The careful planning resulted in the following:

- The real data structure, a doubly-linked list, together with the related functions are hidden from the user of the class.
- The structure of the class makes the implementation of further functions easy.

7.4 IPv6 SIP proxy

Although, the IPv4 SIP proxy is a stateless proxy and IPv6 SIP proxy is also supposed to be stateless, the v6 version has to hold and maintain some information about SIP calls and MEGACO transactions. Furthermore, we had to use threads, because the proxy have to handle parallel MEGACO transactions and SIP calls. The connection between the two main threads (MEGACO and SIP handling) were a queue-like data structure and a state information database.

7.4.1 The functionality of SIP proxies

The v4 proxy is the base of the proxies. It was designed first and then the v6 proxy was built on top of it. Thus the functionality of the v4 proxy is a subset of the v6 proxy. A common feature is that both proxies handle registrations.

All the possible actions of the proxies are presented in Fig 4, where the meaning of the numbers are:

1. Incoming SIP message
2. Registration message
3. Registration response
4. Normal message
5. Message which does not require MEGACO
6. Message which requires MEGACO (pushed into the queue)
7. Message which requires MEGACO (popped form the queue)
8. Message with modified SDP
9. Outgoing SIP message

Note: To be precise, in case of the v4 proxy you can only have actions numbered as: 1,2,3,9 and 4,5 without the TestMegaco function.

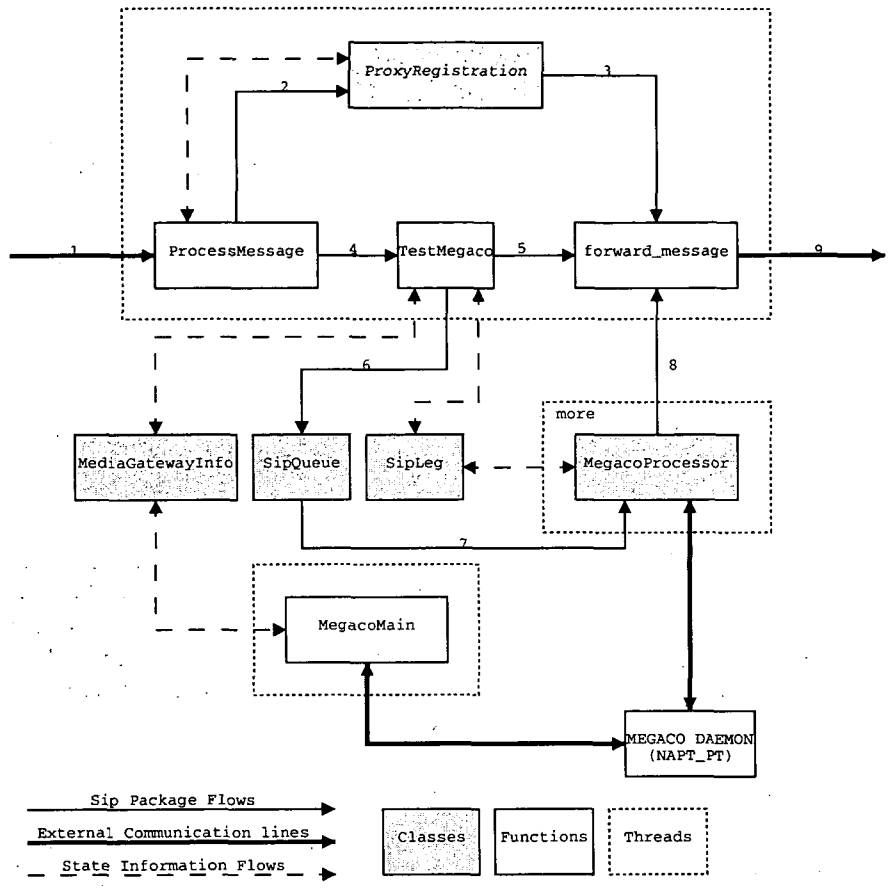

Figure 4: The functionality of SIP proxy on IPv6

7.4.2 Thread handling

Careful synchronization were needed for the reading and writing of the queue (similar to the well-known consumer / producer problem) to prevent incorrect operation and high system load. For blocking the access to data structures, mutexes and signals were used. In case of the queue, threads were put on hold while other processes read or write it, or the queue was empty. As for the state information database, it was enough to block its usage when a thread used it. We implemented some threads to clean up the data structure by erasing the expired data. One thread was necessary for waiting registrations from media gateways (MG), this was separated from the MEGACO thread that handle MEGACO transaction, because the registrations were special transactions.

7.5 Main Restrictions

- No dual-stack implementation.
- SIP server / client and the proxy must be able to handle only unicast connections, no multicast is needed.
- SIP proxy supports UDP only.
- SIP proxy is a stateless-proxy. (It stores data about the IP address conversions and the registration only.)
- Kphone supports UDP only.
- Kphone ensures RTP/RTCP compatibility.

8 Conclusion

Our study is about a novel way of connecting 3G mobile networks based on IPv6 and legacy Internet phones based on IPv4. The main idea is to connect a mobile IPv6 SIP User Agent and another SIP User Agent based on IPv4 via two SIP proxies (IPv6 and IPv4) and NAPT-PT; where the SIP communication between the IPv4 and the IPv6 networks is in fact between the two proxies via NAPT-PT. To control NAPT-PT by the IPv6 SIP proxy MEGACO protocol is used, which has good capabilities but also a highly complex syntax. NAPT-PT plays the role of Media Gateway: it converts packets of the media from IPv4 to IPv6 and vica versa; while the IPv6 SIP proxy corresponds to the Media Gateway Controller: it sends requests to NAPT-PT to reserve or free IPv6 - IPv4 address/port bindings. NAPT-PT also has a DNS-ALG extension to convert DNS queries and responses between the IPv4 and IPv6 networks.

We decided to develop a system demonstrating this communication technique. To do so, first we identified the main parts of the system: the SIP-specific NAPT-PT with DNS-ALG extension, the IPv4- and IPv6-based SIP proxies and the MEGACO daemon. After that we specified, implemented and verified these development units. Several tests were executed to find out the performance of these units, and additionally, the conformance and robustness of our implementation.

We would like to emphasize that at the time of system analysis we did not find any implementation of the previous techniques (although we could use some freely available development libraries), thus we had to develop our own pieces of software. We are still not aware of any open source implementation – which could be used for test purposes – at the time of writing. As far as we know, we are the first, who have analysed such a communication system and created a demonstration version of it.

Our communication environment is a fairly complex system composed of three related yet distinct entities. Thus our software engineering task was not usual: we had to develop parts from kernel to user level, from network to application layer of

the TCP / IP protocol family, in C and C++ languages. Indeed, communication of our software pieces required novel solutions and expert techniques. We could make good use of several development tools, whose roles are emphasized. The contribution of our work is not only the summary of a genuinely new technique, but also comes from the presentation of various software development concepts.

We have several ideas for future research in this topic. Definitely it is worth examining the case of a more general SIP User Agent capable e.g. transporting not only voice but video images thus making a real multimedia IPv6 – IPv4 session.

9 Acknowledgment

This article presents the results of a research work initiated and financed by Nokia Hungary as a joint project between University of Szeged and Nokia Hungary.

We would like to thank Gábor Bajkó, György Wolfner and László Martonossy for drawing our attention to this topic and also for their help in some technical questions. Thanks also to Dénes Bátri for his participation in the parser development and Mihály Bohus for his useful comments. Last but not least, thanks to Gergő Kiss for his valuable participation in the translator development.

10 Appendix: Configuration

10.1 Installation

Kphone is an Internet telephone program, which uses Dissipate, QT and KDE. KDE (stands for K Desktop Environment) is a free desktop system for UNIX-like systems.

Dissipate is a SIP implementation over IPv4. Kphone uses libdissipate to manage media connections. We ported libdissipate and kphone to IPv6, the name of the IPv6 library and program is libdissipate_v6 and kphone_v6.

QT is a cross-platform C++ GUI framework. The creator of QT is Trolltech (www.trolltech.com). There is an edition called 'QT/X11 Free', it is free and has available source code. The QT version 2.2.1 was used for the development, but all versions above 2.2.1 should work properly. All of the QT versions are available via FTP at 'ftp://ftp.trolltech.com/qt/source/'.

10.2 Network environment

The test environment (see Figure 5) contains 5 computers: 2 for only IPv4, 2 for only IPv6 and one for connect the IPv4 and IPv6 networks.

Network topology and IP addresses is described below:

- **v4 client:** IPv4 capable host, SIP UA
- **v4 proxy:** IPv4 capable host, IPv4 SIP proxy and master DNS server of zones '.', 'hu' and 'nokia.hu'

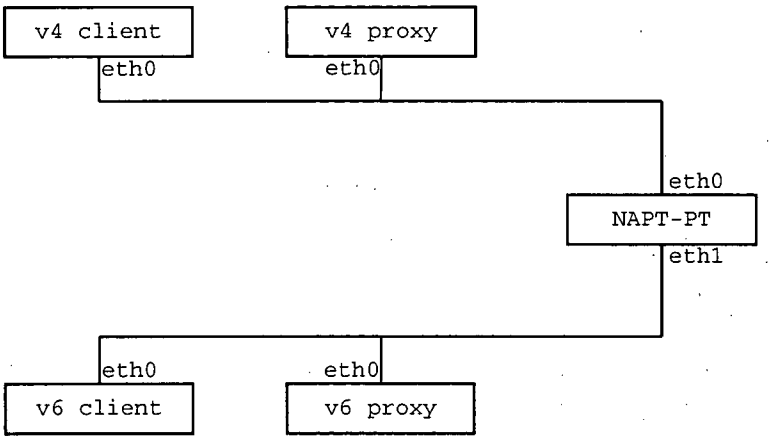

Figure 5: Network topology

- **NAPT-PT:** knows both IPv4 and IPv6 protocols, implements NAPT-PT and runs MEGACO daemon
- **v6 proxy:** IPv6 capable host, IPv6 SIP proxy, master DNS server of zone 'operator.hu' and controls NAPT-PT using MEGACO
- **v6 client:** IPv6 capable host, SIP UA

10.3 NAPT-PT

NAPT-PT is a special router that knows both IPv4 and IPv6 protocols. Because of it, NAPT-PT had to be developed in kernel level. The implementation form of the NAPT-PT is a network device. This means that you can handle it similarly to other network devices, for example the loopback (lo) or an ethernet (ethx) device.

10.3.1 Configuring the 'naptpt' network device

NAPT-PT comes as a kernel patch for Linux v2.2.17. This means you must apply this patch to the kernel. The kernel v2.2.17 can be downloaded from any official kernel mirrors, for example: ftp://ftp.[hu.]kernel.org/pub/linux/kernel/v2.2/linux-2.2.17.tar.gz

In Linux the network devices must be IPv4 and IPv6 addresses assigned to make it processing packets. We suggest that you use private, unused addresses for this purpose.

```
# ifconfig naptpt 10.0.0.1 netmask 255.255.255.255
```

Route IPv4 address pool to 'naptpt' device:

```
# route add -net 192.168.100.0 netmask 255.255.255.0 dev naptpt
```

Do the same with IPv6:

```
# ifconfig naptpt add 3ffe:ffff::1/128
```

Route the IPv4-mapped-IPv6 addresses to 'naptpt' device:

```
# route -A inet6 add ::ffff:0:0/96 dev naptpt
```

Turn on IPv4 and IPv6 packet forwardings:

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

Now, the 'naptpt' device is up and ready to work.

10.3.2 Adding/removing bindings

Adding/removing bindings is basically the task of the MEGACO daemon, but it is possible to add/remove a binding using 'naptconf' utility for testing purposes.

There are two types of bindings, single and double. To add both bindings, you need an IPv6 address/port/protocol triplet, for example: 3ffe:2700:70:1::1/13654/6, where the last component means the transport protocol(6 for TCP, 17 for UDP).

To add a single binding, you must do:

```
# naptconf 5 3ffe:2700:70:1::1 13654 6
```

If the NAPT-PT has unallocated IPv4 address/port pairs, it returns an IPv4 address/port pair, for example 192.168.100.1/6000. This means, if an IPv6 packet is sent through the NAPT-PT which has source address 3ffe:2700:70:1::1, port number 13654 and protocol number 6 (TCP), the NAPT-PT will translate it to IPv4 and the new source address/port will be 192.168.100.1/6000 with the same transport protocol than the original (TCP). The reverse direction: if an IPv4 packet is sent through the translator which has address 192.168.100.1, port 6000 and protocol number 6, it will be translated and the new IPv6 address/port pair will be 3ffe:2700:70:1::1/13654. This is really a binding between address pairs which have a given protocol number:

	addr: 3ffe:2700:70:1::1		192.168.100.1
a given protocol number:	port: 13654	———	6000
	prot: 6		6

To release this binding, you must do:

```
# naptconf 6 3ffe:2700:70:1::1 13654 6
```

Let's see the difference between a single and a double binding: a double one makes two bindings between two address/port pairs, using the previous example:

```
# naptconf 7 3ffe:2700:70:1::1 13654 6
```

The result will be two bindings:

addr: 3ffe:2700:70:1::1		192.168.100.1
port: 13654	———	6000
prot: 6		6

and

addr: 3ffe:2700:70:1::1		192.168.100.1
port: 13655	———	6001
prot: 6		6

The second binding has the almost the same properties than the first one but both IPv4 and IPv6 port numbers are increased by one.

You can add/remove double bindings using the 7/8 functions of naptconf instead of 5/6.

References

[1] J. Postel: Internet Protocol. *RFC 0791* September 1981.

[2] S. Deering and R. Hinden: Internet Protocol, Version 6 (IPv6) Specification. *RFC 2460* December 1998.

[3] C. Huitema: The new Internet Protocol. Prentice Hall 1996.

[4] M. Handley, H. Schulzrinne, E. Schooler and J. Rosenberg: SIP: Session Initiation Protocol. *RFC 2543* March 1999.

[5] G. Tsirtsis and P. Srisuresh: Network Address Translation - Protocol Translation (NAT-PT). *RFC 2766* February 2000.

[6] M. Handley and V. Jacobson: Session Description Protocol (SDP). *RFC 2327* April 1998.

[7] F. Cuervo, N. Greene, A. Rayhan, C. Huitema, B. Rosen and J. Segers: Megaco Protocol 1.0. *RFC 3015* November 2000.

[8] P. Srisuresh, G. Tsirtsis, P. Akkiraju and A. Heffernan: DNS extensions to Network Address Translators (DNS_ALG). *RFC 2694* September 1999.

[9] CVS: Concurrent Versions System, (http://www.cvshome.org/).

[10] Together is the product of TogetherSoft (http://www.togethersoft.com/).

[11] Terence John Parr: Language Translation Using PCCTS and C++ *A Reference Guide*, Automata Publishing Company, San Jose, CA 1993.

[12] Hossam Afifi, Laurent Toutain: Methods for IPv4-IPv6 transition, *The Fourth IEEE Symposium on Computers and Communications* 6 - 8 July, 1999 Red Sea, Egypt.

[13] Alain Durand: Deploying IPv6 *IEEE Internet Computing* Vol. 5, No. 1, February 2001.

[14] ANTLR: Another Tool for Language Recognition, (http://www.antlr.org/).

[15] Gábor Bajkó, Balázs Bertényi, SIP sessions between a 3G network and a SIP-proxy traversing NAT-PT (NOKIA internal report). 2000 August.

[16] Ethereal, (http://www.ethereal.com/).

[17] L. Sógor, L. Martonossy, M. Fidrich, G. Somlai, G. Dikán, P. Hendlein, T. Tarjányi and M. Bohus: Test of inter-working and translation mechanisms between IPv4 and IPv6, *Conference of PhD Students on Computer Sciences*, July 20-23, 2000, Szeged.

[18] Dissipate, a SIP implementation library and Kphone, an internet telephone program (http://www.div8.net/dissipate/).

[19] QT, the crossplatform C++ GUI framework (http://trolltech.com/).

[20] Dimitri van Heesch: Doxygen (http://www.stack.nl/~dimitri/doxygen/).

[21] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson: A Transport Protocol for Real-Time Applications (RTP). *RFC 1889* January 1996.

[22] The Linux Kernel (http://www.linux.org/).

[23] D. Crocker and P. Overell: Augmented BNF for Syntax Specifications (ABNF). *RFC 2234* November, 1997.

[24] R. S. Scowen: Extended BNF - A generic base standard (EBNF). *ISO 14977*

[25] L. Sógor, P. Hendlein, K. Notaisz, M. Fidrich, G. Fóris, G. Kiss, D. Bátri, Gy. Horváth and M. Bohus: Development of a Communication Environment between IPv6 and IPv4 *SZTE Internal Report*, April 2001, Szeged.