Dibbler – a portable DHCPv6 Developer's Guide

Tomasz Mrugalski
thomson(at)klub.com.pl

2006-08-24
0.5.0-alpha-CVS

Contents

1 Intro

Welcome to the Dibbler developer's guide. This document describes various aspects of the compilation and installation of Dibbler server and client. Detailed description of the internal architecture is also provided. People with programming background can find useful informations here. Main purpose of this document is to help contributors to quickly know Dibbler from the inside.

This document is intenteded just as its title states – a guide. It is not a thorough code description. To quickly wander around classes and methods used, see documentation generated with the Doxygen tool (open file `doc/html/index.html`). More informations about documentation is provided in section 4.2.

2 Compilation

Currently Dibbler supports two platforms: Linux with kernels 2.4 and 2.6 series and Windows (XP and 2003). Compilation process is system dependent, so it is described for Linux and Windows separately.

2.1 Linux

To compile Dibbler, extract sources, and type:

```
make client
make server
```

to build client and server. Although parser files are generated using flex and bison++ and those generated sources are included, so there is no need to generate them. To generate it if someone wants to generate it by hand instead of using those supplied versions, here are appropriate commands:

```
make parser
```

to generate client, server and relay parsers.

There occassionaly might be problem with compilation, when different flex version is installed in the system. Proper FlexLexer.h is provided in the SrvCfgMgr and ClntCfgMgr directories.

2.2 Windows

To compile Dibbler under Windows, MS Visual Studio 2003 was used. Project files are provided in the CVS and source archives.

Select project name (server-winxp or client-winxp), click properties, choose ,,Debugging" from ,,Configuration Properties". Adjust ,,Command arguments" to match your directory.

Previous versions were also compiled using MS Visual Studio 2002, but it is not used anymore and is not supported. If you are using MS Visual Studio 2002, there might be a problem with lowlevel-win32.c file compilation. Compiler might complain about missing Ipv6IfIndex in _IP_ADDAPTER_ADDRESSES structure. There is a simple way to bypass this. In
`Program Files/Microsoft Visual Studio.NET/Vc7/PlatformSDK/Include/` directory, there is `IPTypes.h` file. It contains structure:

```
typedef struct _IP_ADAPTER_ADDRESSES {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD IfIndex;
        };
    };
    struct _IP_ADAPTER_ADDRESSES *Next;
    PCHAR AdapterName;
    PIP_ADAPTER_UNICAST_ADDRESS FirstUnicastAddress;
    PIP_ADAPTER_ANYCAST_ADDRESS FirstAnycastAddress;
    PIP_ADAPTER_MULTICAST_ADDRESS FirstMulticastAddress;
    PIP_ADAPTER_DNS_SERVER_ADDRESS FirstDnsServerAddress;
    PWCHAR DnsSuffix;
    PWCHAR Description;
    PWCHAR FriendlyName;
    BYTE PhysicalAddress[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD PhysicalAddressLength;
    DWORD Flags;
    DWORD Mtu;
    DWORD IfType;
    IF_OPER_STATUS OperStatus;
} IP_ADAPTER_ADDRESSES, *PIP_ADAPTER_ADDRESSES;
```

You should slightly modify it. Just add one additional field: `DWORD Ipv6IfIndex;`. Now it should look like this:

```
typedef struct _IP_ADAPTER_ADDRESSES {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD IfIndex;
        };
    };
    struct _IP_ADAPTER_ADDRESSES *Next;
    PCHAR AdapterName;
    PIP_ADAPTER_UNICAST_ADDRESS FirstUnicastAddress;
    PIP_ADAPTER_ANYCAST_ADDRESS FirstAnycastAddress;
    PIP_ADAPTER_MULTICAST_ADDRESS FirstMulticastAddress;
    PIP_ADAPTER_DNS_SERVER_ADDRESS FirstDnsServerAddress;
    PWCHAR DnsSuffix;
    PWCHAR Description;
    PWCHAR FriendlyName;
    BYTE PhysicalAddress[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD PhysicalAddressLength;
    DWORD Flags;
    DWORD Mtu;
    DWORD IfType;
    IF_OPER_STATUS OperStatus;
```

```
    DWORD Ipv6IfIndex;
} IP_ADAPTER_ADDRESSES, *PIP_ADAPTER_ADDRESSES;
```

2.2.1 Flex/bison under Windows

As was mentioned before, flex and bison++ tools are not required to successfully build Dibbler. They are only required, if changes are made to the parsers. Lexer and Parser files (`ClntLexer.*`, `ClntParser.*`, `SrvLexer.*` and `SrvParser.*`) are generated by author and placed in CVS and archives. There is no need to generate them. However, if you insist on doing so, there is an flex and bison binary included in port-winxp. Take note that several modifications are required:

- To generate `ClntParser.cpp` and `ClntLexer.cpp` files, you can use `parser.bat`. After generation, in file `ClntLexer.cpp` replace: `class istream;` with: `#include <iostream>` and `using namespace std;` lines.
- flex binary included is slightly modified. It generates

  ```
  #include "FlexLexer.h"
  ```

  instead of

  ```
  #include <FlexLexer.h>
  ```

  You should add . to include path if you have problem with missing `FlexLexer.h`. Also note that `FlexLexer.h` is modified (std:: added in several places, `<fstream.h>` is replaced with `<fstream>` etc.)

Keep in mind that author is in no way a flex/bison guru and found this method in a painful trial-and-error way.

2.3 DEB and RPM Packages

There is a possibility to generate RPM (RadHat, Fedora Core, Mandrake, PLD and lots of other distributions) and DEB (Debian, Knoppix and other) packages. Before trying this trick, make sure that you have required tools (rpmbuild for RPM;dpkg-deb for DEB packages). Note that this requires root privileges. Package generation is done by the following commands:

```
make release-deb
make release-rpm
```

2.4 Ebuild script for Gentoo

There is also ebuild script prepared for Gentoo users. It is located in the Port-linux/gentoo directory.

2.5 Dibbler in Linux distributions

Dibbler is available in several distributions:

Debian GNU/Linux – use standard tools (apt-get, aptitude) to install dibbler-client, dibbler-server, dibbler-relay or dibbler-doc packages (e.g. apt-get install dibbler-client)

Gentoo Linux – use emerge to install dibbler (e.g. emerge dibbler).

PLD GNU/Linux – use standard PLD's poldek tool to install dibbler package.

2.6 Compilation environment

When compilation is being performed in non-standard envrionment, it is a good idea to examine and modify `Makefile.inc` file. Compiler name, compilation and link options, used libraries and debugging options can be modified there.

2.7 Changing default values

Custom builds might be prepared with different than default compilation options. Here is a list of features, which can be customised:

- Default log level – please modify LOGMODE_DEFAULT define in `Mish/Logger.h`.
- FIXME - describe remaining parameters

2.8 Modular features

In the 0.5.0 release, so called *modular features* were introduced. It is now possible to enable or disable of the Dibbler features. To set, which optional features should be compiled, modify `Makefile.inc` file before starting compilation[1]. Following flags are available:

MOD_CLNT_EMBEDDED_CFG – If this flag is set, client will use hardcoded configuration, instead of reading configuration file. To reasonably use this feature, hardcoded configuration should be modified to match specific needs. See `ClntCfgMgr/ClntCfgMgr.cpp` file for details.

MOD_CLNT_DISABLE_DNSUPDATE – If this flag is set, client will be compiled without DNS Update support, used in FQDN feature. This will make client binary file smaller and will skip the whole poslib library, but client will not be able to perform DNS Updates on its own and will ask server to perform such updates. When DNS Updates are disabled, extra care should be used during server configuration, so all updates will performed on the server side.

MOD_CLNT_BIND_REUSE – Normally it does not make sense to execute server and client on the same machine. It is also not reasonable to execute several client instances on the same host. To prevent such situations, client open normal sockets (without reuse flag). If second client instance is executed, it will fail to create and bind sockets, because required address/port combination is already used by the first instance. However, in some situations this safety check can be unwanted and situation to allow to execute several clients in parallel should be allowed. To allow this, enable flag MOD_CLNT_BIND_REUSE. Note that feature will also make possible to execute server and client on the same node.

MOD_SRV_DISABLE_DNSUPDATE – If this flag is set, server will be compiled without DNS Update support, used in FQDN feature. This will make server binary file smaller and will skip the whole poslib library, but server will not be able to perform DNS Updates on client behalf. According to FQDN standard [10], only server is allowed to execute reverse resolve (PTR record) DNS updates, so in such setup only forward resoling (AAAA record) will be executed be the client.

3 Portability Guide

This section contains guidelines and tips for people intending to port Dibbler to a new achitecture or system. Before attempting to do so, please contact Dibbler author (thomson(at)klub.com.pl) for help. Substantial support will be provided.

[1] In Windows builds, which use MS Visual Studio, those flags must be defined in the project options window

3.1 Low-level System API

To port dibbler to a new system, several of the low level functions have to be implemented. List of those functions is available in Misc/Portable.h file, in section labeled as:

```
/* ********************************************************************* */
/* *** interface/socket low level functions ************************** */
/* ********************************************************************* */
```

Here is a description of the function prototypes:

struct iface * if_list_get() – returns pointer to a list of iface structures. Each structure represents a network interface. This structure is defined in the Misc/Portable.h file. This function should allocate memory for this list.

void if_list_release(struct iface * list) – releases list previously allocated in the if_list_get() function.

int ipaddr_add(const char* ifacename, int ifindex,const char* addr, uint pref, uint valid) – This function adds address specified (in plain text) in addr parameter to the interface named ifacename with interface index ifindex with preferred and valid lifetimes set to pref and valid. Note that some systems might ignore interface name and use ifindex only, or vice versa.

int ipaddr_del(const char* ifacename, int ifindex, const char* addr) – removes address addr (specified in plain text) from the interface ifacename.

int sock_add(char* ifacename,int ifaceid, char* addr, int port, int thisifaceonly, int reuse) – create socket used to read and write data to the ifacename/ifaceid interface, bound to address addr (specified in plain text) and to the port. thisifaceonly parameter specifies if the socket should be bound to the specific interface (1) or not (0). Some systems (e.g. Linux) allow to bind socket in a way that the address/port combination can be bound multiple times. This kind of socket binding allow some advanced tricks like running both server and client on the same host. This parameter is specified by MOD_CLNT_BIND_REUSE, defined (or not) Makefile.inc. This function return file descriptor used to reference to a created socket.

int sock_del(int fd) – delete previously created socket. fd is a file descriptor returned by the sock_add() function.

int sock_send(int fd, char* addr, char* buf, int buflen, int port, int iface) – sends data to addr (defined in packed name)/port, using socket fs. Send buflen byte starting at buf. Send the data using interface iface.

int sock_recv(int fd, char* myPlainAddr, char* peerPlainAddr, char* buf, int buflen) – receive data from the fd socket. Store destination (my) address in a memory located at myPlainAddr, store sender's address in a memory located at peerPlainAddr. The data itself should be stored in a memory located at buf. buflen is a size of a buffer (to avoid buffer overflow). This function returns number of bytes received.

int is_addr_tentative(char* ifacename, int iface, char* plainAddr) – returns information if the address plainAddr added to the ifacename/iface interface is tentative (1) or not (0). It is possible that the Duplicate Address Detection is not yet complete, so other possible return value is inconclusive (2).

Following functions are used to set corresponding parameters, received from the DHCPv6, in the system:

```
int dns_add(const char* ifname, int ifindex, const char* addrPlain);
int dns_del(const char* ifname, int ifindex, const char* addrPlain);
int domain_add(const char* ifname, int ifindex, const char* domain);
int domain_del(const char* ifname, int ifindex, const char* domain);
int ntp_add(const char* ifname, int ifindex, const char* addrPlain);
int ntp_del(const char* ifname, int ifindex, const char* addrPlain);
int timezone_set(const char* ifname, int ifindex, const char*timezone);
int timezone_del(const char* ifname, int ifindex, const char*timezone);
int sipserver_add(const char* ifname, int ifindex, const char*addrPlain);
int sipserver_del(const char* ifname, int ifindex, const char*addrPlain);
int sipdomain_add(const char* ifname, int ifindex, const char*domain);
int sipdomain_del(const char* ifname, int ifindex, const char*domain);
int nisserver_add(const char* ifname, int ifindex, const char*addrPlain);
int nisserver_del(const char* ifname, int ifindex, const char*addrPlain);
int nisdomain_set(const char* ifname, int ifindex, const char*domain);
int nisdomain_del(const char* ifname, int ifindex, const char*domain);
int nisplusserver_add(const char* ifname, int ifindex, const char*addrPlain);
int nisplusserver_del(const char* ifname, int ifindex, const char*addrPlain);
int nisplusdomain_set(const char* ifname, int ifindex, const char*domain);
int nisplusdomain_del(const char* ifname, int ifindex, const char*domain);
```

There are also inet_pton4() (IPv4 address Plain-To-Network), inet_pton6 (IPv6 address Plain-To-Network), inet_ntop4 (IPv4 address Network-To-Plain) and inet_ntop6 (IPv6 address Network-To-Plain) functions, which should be present in the system. If they are not, port-specific part of the dibbler should provide them.

Also function microsleep(int x) should make current process dormant for x microseconds.

An example implementation of those functions, can be found in Port-linux/layer3.c and Port-linux/lowlevel-options-linux.c file. Those files are specific for a Linux system.

To fully port Dibbler, also a main() function must be implemented. It should contain system-specific interface (e.g. registration as a service in Windows environment or detaching to background in Linux "daemon" mode). It is also necessary to include following code in the client implementation:

```
TDHCPClient client(CLNTCONF\_FILE);
client.run();
```

Where CLNTCONF_FILE is a filename of a client configuration file. Similar code should be executed in the server implementation:

```
TDHCPServer srv(SRVCONF\_FILE);
srv.run();
```

See Port-linux/dibbler-client.cpp and Port-linux/dibbler-server.cpp for example implementation, specific to a Linux systems. Implementations for Windows XP are available in the Port-win32 directory.

4 General information

This section covers several loosely related topics.

4.1 Release cycle

Dibbler is being released as a one product, i.e. client, server and relay are always released together. Each version is being designated with three numbers, separated by periods, e.g. 0.4.2. Every time a

new significant functionality is added, the middle number is being increased. When new release contains only fixes and small improvements, only the minor number is changed. Leftmost number is currently set to 0 as not all features mentioned in base DHCPv6 document (RFC3315, [2]) are implemented. When this implementation will be complete, release number will reach 1.0.0. Since DHCPv6 specification is extensive, don't expect this to happen anytime soon.

4.2 Documentation

There are three parts of the documentation: User's Guide, Developer's Guide and a Code documentation. Both guides are written in LaTeX(*.tex files). To generate PDF files, you need to have LaTeXinstalled. To generate Code documentation, a tool called Doxygen is required. All documentation is of course available at Dibbler's homepage.

To generate all documentation, type (in Dibbler source directory):

```
make doc oxygen
```

In this section various common aspects of the Dibbler internal workings are decribed.

4.3 Memory/CPU usage

This section provides basic insight about memory and CPU requirements for the dibbler components.

Folowing paragraphs describe memory and CPU usage measurements. They were taken on a AMD Athlon 2800+ (actual clock speed: 2083MHz), running under Linux 2.6.17.3. Dibbler was compiled by gcc 4.1.2 (exact version number printed by `gcc --version` command:
`gcc (GCC) 4.1.2 20060715 (prerelease) (Debian 4.1.1-9)`).

Every Dibbler component (client, server or relay) is event driven. It means that it does nothing unless some data was received or a specific timeout has been reached. Each component most of the time spends in a `select()` system call. This means that (unless lots of traffic is being received) actual CPU usage is 0. During tests, author was unable to observe any CPU consumption above 0,0%.

In the 0.5.0 release, a compilation options called Modular features was added (see section 2.8). One of the possible way of compiling Dibbler is to disable poslib - a library used to perform DNS Updates. Dibbler binaries compiled without poslib are designated as -wo-poslib. It is possible to compile Dibbler with various compilation options. In particular (enabled by default) `-g` option includes debugging information in the binary file (this greatly affects binary file size, but does not affect memory usage), -O0 (disably any kind of optimisation) or -Os (produce smallest possible code). Debugging informations can be removed using `strip` command (designated below as -stripped). Linux command line tool called `top` was used to measure memory usage. VIRT is a virtual memory size, RES denotes size of actual physical memory used and SHR is a size of a shared memory. See top manual page for details.

VIRT	RES	SHR	%CPU	%MEM	Optim.	filesize	COMMAND
3416	1564	1416	0.0	0.2	-O0	7123510	dibbler-server
3416	1560	1416	0.0	0.2	-O0	751948	dibbler-server-stripped
3328	1544	1400	0.0	0.2	-O0	6533375	dibbler-server-wo-poslib
3328	1548	1400	0.0	0.2	-O0	663592	dibbler-server-wo-poslib-stripped
3220	1436	1292	0.0	0.2	-Os	4596760	dibbler-server run
3140	1424	1276	0.0	0.2	-Os	468776	dibbler-server-wo-poslib
3388	1636	1496	0.0	0.2	-O0	9771605	dibbler-client
3392	1644	1496	0.0	0.2	-O0	725352	dibbler-client-stripped
3296	1608	1472	0.0	0.2	-O0	9183726	dibbler-client-wo-poslib
3300	1612	1472	0.0	0.2	-O0	639240	dibbler-client-wo-poslib-stripped
3212	1472	1336	0.0	0.2	-Os	5901734	dibbler-client-wo-poslib
3120	1456	1320	0.0	0.2	-Os	458984	dibbler-client-wo-poslib

Dibbler stores data internally in lists. This means that server's memory and CPU usage is a linearly proportional to a number of clients it currently supports.

FIXME: Long/performance tests are required.

5 Basic source code informations

This section describes various aspects of Dibbler compilation, usage and internal design.

5.1 Option values and filenames

DHCPv6 is a relatively new protocol and additional options are in a specification phase. It means that until standarisation process is over, they do not have any officially assigned numbers. Once standarization process is over (and RFC document is released), this option gets an official number.

There's pretty good chance that different implementors may choose diffrent values for those not-yet officialy accepted options. To change those values in Dibbler, you have to modify file misc/DHCPConst.h and recompile server or client. Make sure that you build everything for scratch. Use `make clean` in Linux and `Clean up solution` in Windows before you start building a new version.

In default build, Dibbler stores all information in the `/var/lib/dibbler` directory (Linux) or in the working directory (Windows). There are multiple files stored in those directories. However, sometimes there is a need to build Dibbler which uses different directory or filename. To do so, simply edit `misc/Portable.h` file and rebuild everything.

5.2 Memory Manegement using SmartPtr

To effectively fight memory leaks, clever mechanism was introduced. Smart pointers are used to point to all dynamic structures, e.g. messages, options or client informations in server database. Smart pointer will free object by itself, when object is no longer needed. When this is happening? When last smart pointer stops pointing at the object. There is a tradeoff: normal pointers (*) should not be mixed with smart pointers.

Smart pointers are implemented as C++ class templates. Template is called `SmartPtr<TYPE>`.

To quickly explain smart pointers usage, here's short code example:

```
1 void foo()  {
2   SmartPtr<TIPv6Addr> addr = new TIPv6Addr("ff02::1:2");
3   SmartPtr<TIPv6Addr> tmp;
4   if (!tmp) cout << "Null pointer" << endl;
5   tmp = addr;
6   std::cout << addr->getPlain();
7 }
```

What's happened in those lines?

1 – Function starts.

2 – New TIPv6Addr object is created. Smart Pointer (SmartPtr<TIPv6Addr>) is also created to point at this object. Using normal pointer to achive the same goal would look like this:
`TIPv6Addr * addr = new TIPv6Addr("ff02::1:2");`

3 – Another pointer is created. It is equivalent of the classical pointer (TIPv6Addr * tmp).

4 – Simple check if pointer does not point to anything.

5 – Smart pointers can be coppied in a easy way.

6 – Using object pointed by smart pointer is simple

7 – Here magic begins. addr and tmp are local variables, so they are destroyed here. But they are the only smart pointers which access TIPv6Addr object. So they are destroy that object.

In conclusion, object remain in memory as long as there is at least one smart pointer which points to this object. SmartPointers can be easily derefernced. Just add * before them:

```
cout << *addr << endl;
```

SmartPtrs are often used to store various objects in a list. Cool part of this solution is that you can hold objects of various derived classes on one list in a very comfortable manner. There is an additional template defined to create and manipulate such lists. It is called TContainer. There's also useful macro defined to use this without typing too much. Here are two examples how to define list of addresses (both mean exactly the same):

```
TContainer< SmartPtr<TIPvAddr> > addrLst;
List(TIPv6Addr) addrLst;
```

How to use this list? Oh well, another example:

```
1  List(TIPv6Addr) addrLst;
2  SmartPtr<TIPv6Addr> ptr = ...;
3  SmartPtr<TIPv6Addr> tmp;
4  addrLst.clear();
5  addrLst.append(ptr);
6  addrLst.first();
7  tmp = addrLst.get();
8  cout << "List contains " << addrLst.count() << " elements" << endl;
9  addrLst.first();
10 while (tmp = addrLst.get())
11   cout << *tmp << endl;
```

And here is description what that code does:

1 – Address list declaration.

2,3 – SmartPtrs declarations. Just to show variable types.

4 – List can be cleared. All pointers will be destroyed. If they were only pointers to point to some objects, those objects will be destroyed, too.

5 – Append object pointed by ptr to the list.

6 – Rewind list to the beginning.

7 – Get next object from the list. If list is empty or last element was already got, NULL is returned.

8 – An easy way to count elements on the list.

9 – Rewind list to the beginning.

10,11 – A cute example how to print all addresses on the list.

5.3 Logging

To log various informations, Log(LOGLEVEL) macros are defined. There are eight levels of logging:

Emergency – Used to report system wide emergency. Such conditions could not occur in the DHCPv6 client o server, so this logging level should not be used. Called with `Log(Emerg) << "..." << LogEnd`.

Alert – Used to alert an administrator about system wide alerts. This logging level should not be used in DHCPv6. Called with `Log(Alert) << "..." << LogEnd`.

Critical – Used in situations critical to the application, e.g. application shutdown. Fatal errors should be logged on this level. Called with `Log(Crit) << "..." << LogEnd`.

Error – Used to report error situations. For example, problems with binding sockets. Called with `Log(Error) << "..." << LogEnd`.

Warning – Used to report RFC violations, e.g. missing required options, invalid parameters and so on. Called with `Log(Warning) << "..." << LogEnd`.

Notice – Used to report normal operations, e.g. address assignement or informations about received options. Called with `Log(Notice) << "..." << LogEnd`.

Info – Used to report detailed information. DHCPv6 protocol knowledge might be needed to understand those messages. Called with `Log(Info) << "..." << LogEnd`.

Debug – Used to report internal informations. Knowledge about Dibbler source code might be needed to understand those messages. Called with `Log(Debug) << "..." << LogEnd`.

5.4 Names and prefixes

To avoid confussion, various prefixes are used in class and variable names. Class types begin with T (e.g. address class would be named TAddr), enumeration types begin with E (e.g. state enumaterion would be names EState). Dibbler is divided into 4 large functional blocks called managers[2]: address maganger, interface manager, Configuration manager, and transmsission manager. Each of them uses different prefix: Addr, Iface, Cfg or Trans. There are also objects shared among them: messages (Msg prefix) and options (Opt prefix). Often there are two derived versions: related to client (Clnt prefix) or related to server (Clnt). Rel prefix is used to denote Relay related classes. Here are examples of some class names:

TAddrMgr – Address manager, common version.

TClntAddrMgr – Address manager, client version.

TAddrIface – Interface representation, used in address manager.

TAddrAddr – Address representation used in address manager.

TSrvIfaceMgr – Interface manager, server version.

TClntIfaceIface – Interface representation used in client interface manager.

TClntMsg – Message represented on the client side.

TClntOptPreference – Prefernce option used on the client side.

TIfaceSocket – Socket used in the interface manager.

[2] They are described in the following sections of this document

TClntCfgAddr – Address used in the client config manager.

Also note that class function names start with small letters (e.g. `bool TOpt::isValid();`) and class variables start with capital letters (e.g. `bool TOpt::IsValid;`).

5.5 Configuration file parsers

Note: Similar approach is used in server, client and relay. In following section when reference to a specific file is needed, client files are used. To find corresponding files related to server and relay, substitute `Clnt` with `Srv` or `Rel`.

Dibbler uses standard lexer/parser. Lexer is generated using flex. Parser is generated with bison++ (full source code for bison++ is provided with Dibbler sources). See `ClntCfgMgr/ClntParser.y` and ClntCfgMgr/ClntLexer.l for details. Make sure that you have flex installed (bison++ is provided with the dibbler source code). To generate parser and lexer code, type:

```
make bison (just once, to compile bison++)
make parser (each time you modify *.l or *.y files)
```

5.5.1 Parsing

Configuration file reading is done using Flex and bison++ tools. Flex is so called lexer. Its responsibility is to read config file and translate it into stream of tokens. [3] For example, this config file:

```
iface eth0 {
  class { pool 2000::1-2000::9 }
}
```

would be translated to following stream of tokens: [IFACE] [STRING:eth0] [] [CLASS] [] [POOL] [ADDR:2000::1] [-] [ADDR:2000::9] [] []. This stream of token is then passed to parser. This parser is generated by bison++. Parser checks if that particular sequence of tokens makes sense. In this example, interface object will be created, which contains one class object, which contains one pool.

Is is sometimes very useful to define some parameter, usually associated with some level, on higher scope level. For example, if there are 3 classes, instead of defining the same valid-lifetime value on each of them, that parameter may be defined on the interface level or even at the top level. This is important to remember during parsing. Each subsequent element must inherit its parent properties (class object must inherit parameter values defined on the interface level).

To accomplish this feat, simple stack was implemented. For example, in server parser, following methods are called before and after interface definitions.

```
void SrvParser::StartIfaceDeclaration()
{
    // create new option (representing this interface) on the parser stack
    ParserOptStack.append(new TSrvParsGlobalOpt(*ParserOptStack.getLast()));
    SrvCfgAddrClassLst.clear();
}

bool SrvParser::EndIfaceDeclaration()
{
    // create and add new interface to SrvCfgMgr
    ...
    // remove last option (representing this interface) from the parser stack
```

[3] To be precise, Flex generates lexers, so it should be called lexer generator.

```
    ParserOptStack.delLast();
    return true;
}
```

5.5.2 Using parsed values

Lexer and parser are created in the Client Configuration Manager. See ClntCfgMgr/ClntCfgMgr.cpp. Following code is executed in the ClntCfgMgr constructor[4]

```
yyFlexLexer lexer(\&f,\&clog);
ClntParser parser(\&lexer);
result = parser.yyparse();
matchParsedSystemInterfaces(\&parser);
validateConfig();
```

f and clog are normal C++ ifstream and ofstrem objects, associated with configuration file or a standard output. Configuration file is passed to the constructor of the entire TDHCPClient object, which is usually located in the main() function.

Example mentioned above works as follows:

- Read all interfaces from the system (using System API). This is done in Interface Manager and is not important right now.
- Create lexer object (it will read configuration file and convert it into stream of tokens)
- create parser, which will interpret stream of tokens.
- Match interfaces present in system with those specified in the configuration file.
- Validate configuration file to check if there are no logical errors, like T1¿T2, specified both stateless and request for ia, etc.

5.5.3 Embedded configuration

Note: This feature applies to the client only.

Another way of defining client configuration was introduced in the 0.5.0 release. Instead of reading configuration file, configuration can be hardcoded in the binary file itself. See MOD_CLNT_EMBEDDED_CFG flag description in section 2.8.

6 Architecture

General architecture is common between server, client and (to some extent) relay. In all cases, classes are divided into several major groups:

IfaceMgr – Interface Manager. It represents all network interfaces present in the system. They're represented by TIfaceIface objects and stored in IfaceLst. Each interface has list of open sockets, represented with TIfaceSocket objects. There are also a number of auxiliary functions for getting proper interface. IfaceIface objects also provide methods to add, update and remove addresses.

AddrMgr – Address Manager. It is an address database, which stores all informations about clients, IAs and associated addresses.

[4] Actual code is much more complicated, but unnecessary lines were removed for a clarification reasons.

CfgMgr – Config Manager. It is being used to read configuration information from config file and provide those informations while runtime. Common mechanisms shared between server and client are scarce, so this base class is almost empty.

TransMgr – Transmission Manager, sometimes called Transaction Manager. It is responsible for network interaction and core DHCPv6 logic. It sends various messages when such need arise, matches received responses with sent messages, retransmits messages etc. It contains list of messages currently being trasmitted.

Messages – There is one parent class of all messages. It contains several basic functionalites common to all messages.

Options – There are multiple option classes. Note that some classes are designed to represent one specific option (e.g. OptIAAddress) and other are not (e.g. OptAddrLst can contain address list, so it can be used as DNS Resolvers, SIP servers o NIS servers option).

Misc – This cathegory (or rather directory) contains various miscellanous classes and functions.

None of those classes is used directly. Client, server and relay uses derived classes.

They are all created within DHCPClient or DHCPServer objects in client or server, respectively. DHCPRelay object will perform similar function for relays.

6.1 Client Architecture

Client is represented by a DHCPClient object. It contains 4 large managers, each with its own functions. Also messages and options are defined:

TClntIfaceMgr – contains client version of the IfaceMgr. Major difference is a TClntIfaceIface class, an enhanced version of the IfaceIface. It provides methods to set up various options on the physical interface. Those methods are used by Options representing options.

TClntAddrMgr – Client version supports additional, client related functions, e.g. tentative timeout used in DAD procedure. It also simplifies database handling as there will always be only one client in the database.

TClntCfgMgr – Client related parser. TClntCfgMgr and related objects are designed to provide easy access to parameters specified in the configuration file. ClntCfgIface is a very important class as most of the parameters is interface-specific.

TClntTransMgr – Core logic of the Client. It uses all other managers to decide what actions should be taken at occuring circumstances, e.g. send REQUEST when there are less addresses assigned than specified in the configuration file.

TClntMsg – All messages have client specific classes. Those objects are created as new messages are being sent. After server message reception, object is also created and passed to the original message. For example, client sends **SOLICIT** message and server send **ADVERTISE** message. Reply will be passed by invoking `answer(msgAdvertise)` method on the `msgSolicit` object.

TClntOpt – There are client specific options defined. Each of those options has `doDuties()` method which is called if this option was received in a proper reply message from the server. It calls appropriate methods in TClntIfagrMgr which set specific options in the system.

6.2 Server Architecture

Server is represented by a DHCPServer object. It contains 4 large managers, each with its own functions. Also SrvMessages and SrvOptions are defined:

TSrvIfaceMgr – contains server version of the IfaceMgr. There are almost no modificiation compared to common version.

TSrvAddrMgr – Client version supports additional, client related functions, e.g. tentative timeout used in DAD procedure. It also simplifies database handling as there will always be only one client in the database.

TSrvCfgMgr – Client related parser. TSrvCfgMgr and related objects are designed to provide easy access to parameters specified in the configuration file. SrvCfgIface is a very important class as most of the parameters is interface-specific.

TSrvTransMgr – Core logic of the client. It uses all other managers to decide what actions should be taken at occuring circumstances, e.g. send REQUEST when there are less addresses assigned than specified in the configuration file.

TSrvMsg – Server version of the messages. Each time server receives a message, TSrvMsg is created. Depending of its type, TSrvAdvertise of TSrvReply message is created. As parameter to its contructor original message is passed. After creating message, it is sent back to the client and stored for possible retransmission purposes.

TSrvOpt – Server version of the Option representing objects. They are just used to store data, so they are considerably simpler than client versions.

6.3 Relay Architecture

Preliminary relay version was available in the 0.4.0 release. It consists of serveral simple blocks:

TRelIfaceMgr – contains relayr version of the IfaceMgr. There are almost no modificiation compared to common version, execept decodeMsg() and decodeRelayRepl() methods.

TRelCfgMgr – Relay related parser. TRelCfgMgr and related objects are designed to provide easy access to parameters specified in the configuration file. RelCfgIface is a very important class as most of the parameters is interface-specific.

TRelTransMgr – It's plain simple manager. It's only function is to relay received message on all interfaces.

TRelMsg – From the relay's point of view, all messages fall to one of 3 categories: Generic (i.e. not encapsulated) messages, RelayForw (already forwarded by some other relay) and RelayRepl (replies from server). Most of the messages is threated as generic message.

TRelOpt – Similar approach is used to handle options. Expect RELAY_MSG option (which contains relayed message) and interface-id option (which contains identifier of the interface), all options are threated as generic options, which are handled transparently.

7 FAQ

This section describes various Dibbler aspects.

XML files – After performing any action, server, client and relay store their internal information into XML files. As for 0.4.1 version, those files are never read, just written. This feature can be used as a debugging tool. However, it's main purpose is the ability to process and present internal state in some external form. For example using with css styles or after processing via XSLT parsers, server statistics can be presented as a web page.

Message building – Each TMsg object (see Messages/Msg.h) has Options list. Options (TOpt derived objects) are created (usually in the constructor). They're stored as objects. For good example, see appendRequestedOptions() method in the client messages (ClntMessages/ClntMsg.cpp). Each option and message has method storeSelf(), which is called just before message is being sent.

You might ask: what about retransmissions? Message is built each time it is being resent. That might seem inefficient, but there is one option called Estimated. It specifies how long does this particular transaction is being processed. So each time retrasmission is in fact a slightly different message. It differs in that option, so UDP checksum is different, so it has to be rebuilt.

8 Tips

- Linux: Running client and server on the same host requires client recompilation with specific option enabled. Please edit `misc/Portable.h` and set `CLIENT_BIND_REUSE` to `true`. This will allow to receive data from local server, but will also disable checking if there is another client running. So you can run multiple clients, which is a straight road to trouble. You were warned.

- Ethereal, a widely used network sniffer/analyzer has a bug with parsing DHCPv6 message: SIP options are always reported as malformed. Also NIS/NIS+ options have improper values (not comformant to RFC3898). To work around that problem, download packet-dhcpv6.c from Dibbler homepage and recompile Ethereal. Dibbler's author sent patches to the Ethereal team. Those changes should be included in the next Ethereal release. **NOTE:** This is no longer true. Patch was accepted and now Ethereal prints informations properly.

- If you are reading this Developer's Guide, then Hey! You're probably a developer! If you found any bugs (or think you found one), go to the http://klub.com.pl/bugzilla and report it. If your report was a mistake – oh well, you just lost 5 minutes. But if it was really a bug, you have just helped improve next Dibbler version.

- If you have any questions about Dibbler or DHCPv6, feel free to mail me, preferably via Dibbler mailing list. All links are provided on the project website.

References

[1] S. Thomson, and T. Narten "IPv6 Stateless Address Autoconfiguration", RFC2462, IETF, December 1998

[2] R. Droms, Ed. "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC3315, IETF, July 2003

[3] H. Schulzrinne, and B. Volz "Dynamic Host Configuration Protocol (DHCPv6) Options for Session Initiation Protocol (SIP) Servers", RFC3319, IETF, July 2003

[4] S. Thomson, C. Huitema, V. Ksinant and M. Souissi "DNS Extensions to Support IP Version 6", RFC3596, IETF, October 2003

[5] O. Troan, and R. Droms "IPv6 Prefix Options for Dynamic Host Configuration Protocol (DHCP) version 6", RFC3633, IETF, December 2003

[6] R. Droms, Ed. "DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC3646, IETF, December 2003

[7] R. Droms, "Stateless Dynamic Host Configuration Protocol (DHCP) Service for IPv6", RFC3736, IETF, April 2004

[8] V. Kalusivalingam "Network Information Service (NIS) Configuration Options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC3898, IETF, October 2004

[9] S. Venaas, T. Chown, and B. Volz "Information Refresh Time Option for DHCPv6", work in progress, IETF, January 2005

[10] B. Volz "The DHCPv6 Client FQDN Option", work in progress, IETF, September 2005