Dibbler – a portable DHCPv6
Developer's Guide

Tomasz Mrugalski
thomson(at)klub.com.pl

2005-03-16

0.4.0

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 the following sections.

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:

```
cd ClntCfgMgr
make parser
```

to generate client parser, and:

```
cd SrvCfgMgr
make parser
```

to generate server parser.

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 meet your directory.

If you are using MS Visual Studio 2003, 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 PLD GNU/Linux distributions. Author also performs necessary steps to include Dibbler in Gentoo and Debian GNU/Linux distributions.

3 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 Basic informations

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

4.1 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.

4.2 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.

4.3 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`.

4.4 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[1]: 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.

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;`).

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

5 Common Architecture

General architecture is common between server and client. In both cases, all 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.

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 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.

7 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.

8 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.

9 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.
- 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 you report was a mistake – oh well, you just lost 5 minutes. But if it was really a bug, you have just improved next Dibbler version.
- If you have any questions about Dibbler or DHCPv6, feel free to mail me.