The ControlProxy Manual

Jelmer Vernooij

12th March 2004

Abstract

This document is still a work-in-progress. We're doing our best to keep it up-to-date and understandable, but please don't hesitate to contact us if you have comments or questions.

All comments, questions and updates are welcome at jelmer@vernstok.nl <mailto:jelmer@vernstok.nl>.

Chapter 1

Introduction

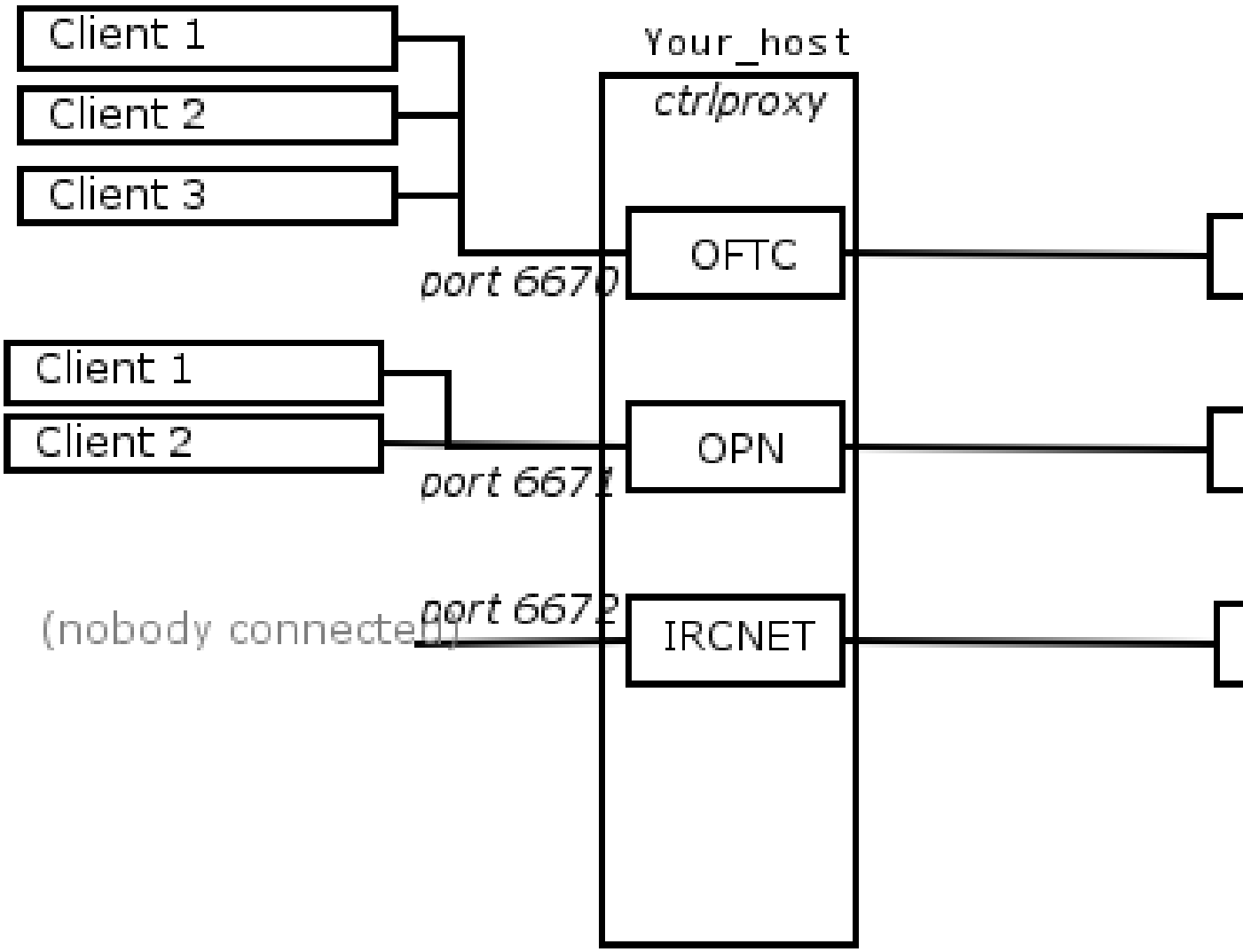
1.1 Why ctrlproxy?

CtrlProxy is a project I started because I got bored with running irssi in screen on my server. My server isn't very fast and that meant when it was on high load ircing was getting pretty hard. I could of course run irssi on my workstation, but my workstation isn't on 24/7 and some people depend on the channel logs I generate.

The structure of ctrlproxy is very modular and it is easily extendible.

1.2 What is ctrlproxy?

Ctrlproxy is a so-called IRC proxy or BNC (bouncer). It keeps a permanent connection to one or more IRC servers. The user can then connect and disconnect his/her IRC client to the bouncer without actually disconnecting from the 'real' IRC server.



1.3 Features

- Connect to one server with many clients under one nick transparently
- Connect to multiple servers using only one process
- CTCP support when no client is attached
- irssi-style logging support
- Transparent detaching and attaching of clients
- Password support
- Replication support (from memory)
- Auto-Away support
- Keeping track of events occuring
- Direct, inetd-style interfacing with local IRC servers (such as bitlbee)
- Responses to queries are only sent to the originator of the query
- SSL/GNUTLS support

1.4 Requirements

- libpopt
- GNU glib
- libxml2

Some of the modules have additional requirements. Read the chapters about those specific modules for details.

Part I

Installation

Chapter 2

Installation

2.1 Precompiled packages

Some distributions come with a packaged version of ctrlproxy. Wilmer van der Gaast <mailto:lintux@lintux.cx> is maintaining the debian ctrlproxy package and Aron Griffis <mailto:agriffis@gentoo.org> maintains the gentoo package of ctrlproxy.

ctrlproxy is included with the BSD ports collection.

If you already have a packaged version of ctrlproxy installed, you can skip this chapter.

2.2 Getting the source code

The source of ctrlproxy can be downloaded from the ctrlproxy homepage <http://ctrlproxy.vernstok.nl/>. The source files available there can be unpacked using tar and gzip:

```
$ tar xvgz ctrlproxy-2.7.tar.gz
ctrlproxy-2.7/AUTHORS
...
```

If you wish to use the bleeding-edge version of ctrlproxy, you can download the sources from CVS.

2.2.1 Downloading from CVS

Ctrlproxy CVS can be accessed by doing:

```
$ cvs -d :pserver:anonymous@cvs.vernstok.nl:/cvs login
```

(when asked for a password, press enter)

```
$ cvs -d :pserver:anonymous@cvs.vernstok.nl:/cvs co -r UNSTABLE ctrlproxy
ctrlproxy
ctrlproxy/AUTHORS
ctrlproxy/README
...
```

Make sure you run aclocal, autoheader and autoconf in the source directory (`ctrlproxy/`) so that the `configure` script is generated correctly.

NOTE

You have to use at least autoconf/autoheader 2.50!

2.3 Compiling from source

First, run the `configure` script:

```
$ ./configure
```

If this script does not detect all libraries and headers, while they are present, specify the locations using command line arguments to `configure`. Run **./configure –help** for details.

After `configure` has finished, run **make**.

Now that ctrlproxy has been built, find your system administrator or become root yourself and (get him/her to) run **make install**.

Part II

Configuration

Chapter 3

Configuration file syntax

Ctrlproxy uses a XML as it's RC file. The syntax of XML files is described much better in other documents on the web and is beyond the scope of this document.

Take a look at the `ctrlproxyrc.example` file that is distributed with ctrlproxy. It should give you a good impression of what a ctrlproxyrc file is supposed to look like.

The root element contains 2 elements: plugins and networks. These are discssed below.

3.1 Plugins

Contains various <plugin> elements, which each represent a plugin that can be loaded. When the autoload attribute is set, the plugin will be loaded when ctrlproxy starts.

The `file` attribute is required and should specify either an absolute path to a plugin or the name of a plugin in the default modules dir (something like `/usr/lib/ctrlproxy`).

The <plugin> element should contain plugin-specific elements. See the documentation for the individual plugins for details.

3.2 Networks

The <networks> element contains several <network> elements, each representing an IRC network.

Attributes that can be specified on a network element are:

name Name of the network. Something like "OPN", "OFTC" or "IRCNet". The name of the first server is used if this is not specified.

client_pass Password a client should use to authenticate when it connects. Defaults to empty string, in which case authentication will be disabled.

nick Initial nick name to use on this network. Defaults to UNIX user name.

username User name to report in hostmask. Defaults to UNIX user name.

ignore_first_nickchange IRC clients always send a NICK command to the IRC server after they have connected. Ctrlproxy happily passes this new nick name on to the real server. If you want ctrlproxy to ignore the first nick change that a client sends, add this attribute.

fullname Full name to report (for example in **/WHOIS** information). Defaults to the full name specified in the gecos field of your NSS passwd backend (usually the file `/etc/passwd`.

autoconnect Specifies whether to connect to this network at start-up.

3.2.1 Listeners

Clients need to be able to connect to ctrlproxy. This is done using so-called 'listeners'. The element <listen> can contain several elements from transports that ctrlproxy should listen on.

For a description of the configuration of the various avaiable transports that can be used for listening, read their chapter in modules part.

Example:

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="socket"/>
   </plugins>
   <networks>
      <network name="OPN" autoconnect="1">
         <listen>
            <ipv4 port="6667"/>
         </listen>
      </network>
   </networks>
</ctrlproxy>
```

3.2.2 Channels

A <network> element can also contain several <channel> elements. Each channel should have a "name" attribute which should contain the name of the channel.

The "autojoin" attribute is voluntary and specifies whether the channel should be joined automatically when ctrlproxy connects to the network.

Example:

```
<ctrlproxy>
   <networks>
      <network name="OPN">
         <channel name="#samba"/>
         <channel name="#samba-technical" autojoin="1"/>
      </network>
   </networks>
</ctrlproxy>
```

3.2.3 Servers

Similar to the <listen> element is the <servers> element. It contains possible transport configuration that is used to connect to the network.

Note that ctrlproxy always only connects to exactly *one* server at once. It starts by connecting to the first server and tries the others in the list if that one fails.

Again, see the documentation for the specific transport plugins for details.

Example:

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="socket"/>
```

```
    </plugins>
    <networks>
       <network name="OPN" autoconnect="1">
          <servers>
             <ipv4 host="irc.freenode.net"/>
             <ipv6 host="irc.ipv6.freenode.net"/>
             <ipv4 host="irc.nl.linux.org"/>
          </servers>
       </network>
    </networks>
 </ctrlproxy>
```

3.2.4 Autosend

A network element can contain one or more <autosend> elements. These should contain raw IRC commands that are sent to the server after ctrlproxy has connected to it.

Example

```
<ctrlproxy>
   <networks>
      <network name="OPN">
         <autosend>PRIVMSG nickserv :identify mysecretpassword</autosend>
         <autosend>PRIVMSG ctrlsoft :Hi! I'm using ctrlproxy!</autosend>
      </network>
   </networks>
</ctrlproxy>
```

3.3 Linestacks

Linestacks are a system used inside ctrlproxy for storing lines of IRC data. Since this data can be stored in various places, and everybody has different needs, it is possible to use a different linestack then the default one.

Since some backends need additional information (such as a filename to store the data in, or a database name), you can specify an additional argument to the linestack backend.

If you specify the attribute *linestack* of the element network you can override the default linestack backend.

The attribute *linestack_location* can be used to override the setting of the argument to the linestack backend.

See the documentation on a specific backend for information on setting the default argument, if applicable.

By default, the linestack that is first loaded is used.

See the documentation about the various available linestack backends for more information.

3.4 Replication

Replication (short for 'backlog replication') is the system that stores certain IRC lines and then sends them to the user at a certain moment.

Chapter 4

Frequently Asked Questions

4.1 Connecting from XChat

Question: I have set up ctrlproxy, but nothing happens when I connect to it via the XChat client.

Answer: Ctrlproxy needs a password. Use the following /server command : **/server hostname port password**.

4.2 Supported operating systems

Question: On what operating systems does ctrlproxy run?

Answer: It has been tested with all major Linux distributions and FreeBSD. However, it should run on pretty much every OS that is POSIX-compatible.

4.3 Supported IRC clients

Question: What IRC clients does ctrlproxy work with?

Answer: All IRC clients should be supported (ctrlproxy acts just like any other IRC server and follows the RFC's). Please contact the mailinglist if you find that a client is not working well together with ctrlproxy.

Part III

Modules

Chapter 5

chapter

Remote administration

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

5.1 Description

This module provides a simple interface for remote administration of ControlProxy. Commands can be executed by either using the /CTRLPROXY command in your IRC client or sending them to the nick `ctrlproxy` on a network.

The syntax for the commands is very simple: the command should be followed by one or arguments, seperated by spaces. Quoting is not supported.

5.2 Commands

The following commands are supported:

ADDNETWORK <**name**> Adds a new network with the specified name.

ADDLISTEN <**network**> <**type**> [<**key**>=<**value**>] **[...]** Adds a new 'listener' to the specified network with the specified type and options.

Example: **addlisten OPN ipv4 port=6676**

ADDSERVER <**network**> <**type**> [<**key**>=<**value**>] **[...]** Adds a new server to the specified network with the specified type and options.

Example: **addserver OPN ipv4 host=irc.freenode.net**

CONNECT <**network**> Connect to the specifed network. Ctrlproxy will connect to the first known server for this network.

DELNETWORK <**network**> Remove the specified network. The network may not be connected.

DIE Disconnect all clients and servers and exit ctrlproxy.

DISCONNECT <**network**> Disconnect from the specified network.

DETACH Detach client from the proxy.

LISTNETWORKS Prints out a list of all networks ctrlproxy is connected to at the moment.

NEXTSERVER Makes the specified network disconnect from the current server and go to the next one.

LOADMODULE <**location**> Load DSO module (aka 'plugin') from the specified location.

RELOADMODULE <**location**> Reload the DSO module at the specified location. This does the same as doing a **UNLOADMODULE** followed by a **LOADMODULE**.

UNLOADMODULE <**location**> Unload the DSO module which was loaded from the specified location. This may or may not work correctly, depending on the plugin you are trying to unload.

LISTMODULES Prints out a list of all currently loaded plugins.

DUMPCONFIG Prints out the current configuration file XML data.

SAVECONFIG Save the (updated) XML configuration file to the location it was loaded from (usually `$HOME/.ctrlproxyrc`).

HELP Prints out list of available commands.

5.3 Example commands

Adding a new network called 'OFTC', listening for incoming connections on port 6667.

```
ADDNETWORK OFTC
ADDSERVER OFTC ipv4 host=irc.oftc.net
ADDLISTEN OFTC ipv4 port=6667
CONNECT OFTC
```

5.4 Configuration

The following XML elements are supported:

without_privmsg If this element is present, you can not execute admin commands by sending them to the nick 'ctrlproxy'. This can be useful for IRC clients that don't allow unknown IRC commands to be executed.

Chapter 6

chapter

Automagic away

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

6.1 Description

This module sets your IRC status to 'away' after you have been inactive('idle') for a certain period of time.

6.2 Configuration

The following XML elements are supported:

message Message to set AWAY mode to when idle for too long.

time Number of seconds you have to be idle before setting AWAY.

only_noclient If set, auto-away will only set AWAY when there are no clients connected to ctrlproxy.

6.3 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="auto-away">
         <message time="600">I've been idle for 10 minutes, so I'm probably away
      </plugin>
      <plugin autoload="1" file="socket"/>
   </plugins>
```

```
   <networks>
      <network name="OFTC">
         <servers><ipv4 host="irc.oftc.net"/></servers>
         <channel name="#flood.nl" autojoin="1"/>
      </network>
   </networks>
</ctrlproxy>
```

Chapter 7

chapter

Standard CTCP module

Version: 0.1

Author: Jelmer Vernooij <mailto:>

Homepage: <http://jelmer.vernstok.nl/ctrlproxy/>

7.1 Description

Simple CTCP module that implements some basic CTCP commands. Use for this module is having CTCP support available when there is no client connected that can answer CTCP queries and providing the ability to detect ctrlproxy.

The following CTCP commands are supported:

VERSION
TIME
FINGER
SOURCE
CLIENTINFO
PING

Chapter 8

chapter

Irssi-style log files

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

8.1 Description

Module that logs IRC data to the specified file in the same format that the irssi(1) IRC client uses.

Each channel or nick gets it's own seperate log file, which is located in a directory with the name of the IRC network.

If no directory is specified, data will be logged to *$HOME/.ctrlproxy/log_irssi/$NETWORK/$CHANNEL*.

8.2 Configuration

The following XML elements are supported:

logfile Should specify a base path that log files are to be generated in. For each network, a subdirectory will be created in this directory.

8.3 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="log_irssi">
         <logfile>/home/jelmer/log/ctrlproxy</logfile>
      </plugin>
      <plugin autoload="1" file="socket"/>
   </plugins>

   <networks>
      <network name="OFTC">
         <servers><ipv4 host="irc.oftc.net"/></servers>
```

```
            <channel name="#flood"/>
        </network>
    </networks>
</ctrlproxy>
```

Chapter 9

chapter

'No' replication

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

9.1 Description

This is the module you should use when you do not want any replication of what has been said on a channel or in a query, but you do want to know what channels you are on.

Chapter 10

chapter

Very simple replication

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

10.1 Description

This module adds very simple replication. All lines that were sent after the last line that has been said by the user are stored and sent to a client that connects to ctrlproxy.

Chapter 11

chapter

Replicate lines matching certain patterns

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

11.1 Description

Rather simple replication module that works similar to the repl_simple module, but only replicates the lines matching certain patterns.

11.2 Configuration

The following XML elements are supported:

match Log all lines that contain the string this element contains.

11.3 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="repl_highlight">
         <match>jelmer</match>
         <match>ctrlproxy</match>
      </plugin>
   </plugins>
</ctrlproxy>
```

Chapter 12

chapter

Replicate lines on demand

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

Requirements: admin module

12.1 Description

Replicates certain lines of backlog, when the user requests so via a **/CTRLPROXY** or **/MSG CTRLPROXY** command.

12.2 The BACKLOG command

The **BACKLOG** command can be run by using the command interface of the 'admin' module.

Without any arguments, the BACKLOG command replicates all the backlogs for the current channel.

WIth one argument, the name of a channel, all lines on that channel are replicated.

With two arguments, the first argument contains the name of a channel and the second the number of lines to replicate, the specified number of (most recent) lines from the specified channel is replicated.

12.3 Example commands

```
BACKLOG
BACKLOG #flood.nl
BACKLOG #linux.nl 500
```

Chapter 13

chapter

Very lastdisconnect replication

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

13.1 Description

This module adds very simple replication. All lines that were sent after the last time you disconnected a client are stored and sent to a client that connects to ctrlproxy.

Chapter 14

chapter

Support for IPv4, IPv6 and pipes

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

14.1 Description

This module provides support for connecting to remote servers using IPv4, IPv6 and unix pipes, as well as listening for client connections using these connection types.

As this module is currently the only module providing connection support, it is essential for basic use of ctrlproxy.

Connecting or listening using SSL over IPv4 or IPv6 is supported when a SSL library was found at configure time.

When acting as a SSL server (e.g. waiting for connections from clients and communicating with them using SSL), ctrlproxy needs to have a certificate file and a private key file. This can be generated using the `mksslcert.sh` script distributed with ctrlproxy.

14.2 Configuration

The following XML elements are supported:

sslkeyfile Name of file to load private SSL key from. Only required when acting as a server

sslcertfile Name of file to load certificate from. Only required when acting as a server

tos ***sslcafile*** Name of file that contains the CA file to load. Only required when acting as a server and when GNUTLS is used.

value TOS value for outgoing packages. Has to be 'Minimize-Delay', 'Maximize-Throughput', 'Maximize-Reliability', 'Minimize-Cost' or 'Normal-Service' (default)

14.3 Configuration

After this module is loaded, the following three new elements are supported in <listen> and <servers>:
ipv4
ipv6
pipe
ipv4 and ipv6 support the following attributes:

ssl Enable SSL

host Host name or IP address to connect to.

port Port to connect to or listen on.

When connecting, the pipe element can contain one member element <path> and several <arg> elements. These should contain a program with arguments to execute.

In listen mode, a file attribute (attribute, not element!) should be specified, containing the file name of the unix socket to create. If no file name is specified, one will be generated.

14.4 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="socket">
         <sslcertfile>ctrlproxy.pem</sslcertfile>
         <sslkeyfile>ctrlproxy.pem</sslkeyfile>
      </plugin>
   </plugins>
   <networks>
      <network name="BEE">
         <servers>
            <pipe>
               <path>/usr/sbin/bitlbee</path>
            </pipe>
            <ipv4 host="localhost"/>
         </servers>
         <listen>
            <ipv4 ssl="1" port="6667"/>
      </network>
      <network name="DSR">
         <servers>
            <ipv6 host="irc.ipv6.distributed.net"/>
            <ipv4 host="irc.distributed.net" port="994" ssl="1"/>
         </servers>
         <listen>
            <ipv4 port="6668"/>
            <ipv6 port="6669" ssl="1"/>
         </listen>
      </network>
   </networks>
</ctrlproxy>
```

Chapter 15

chapter

Logging ctrlproxy messages via IRC

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

15.1 Description

This module sends all ctrlproxy log files to all connected clients using IRC NOTICE's.

Currently only messages from the main ctrlproxy process will be send as NOTICE's. Messages from plugins will be ignored.

Chapter 16

chapter

Strip query answers for other clients

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

16.1 Description

One problem with ctrlproxy's multi-client support is the fact that when one client does a query (such as a WHOIS), all other clients get the answer. This module fixes that problem.

The following queries are intercepted by this module:

- WHOIS
- WHO
- NAMES
- LIST
- TOPIC
- WHOWAS
- STATS
- VERSION
- LINKS
- TIME
- SUMMON
- USERS
- USERHOST
- ISON

Chapter 17

Custom logging module

Logging in a predefined format

Version: 0.1

Author: Jelmer Vernooij <mailto:>

Homepage: <http://jelmer.vernstok.nl/ctrlproxy/>

17.1 Description

Module that writes logs to one or more files using a defined format.

This module may be used to write out log files that can be parsed by scripts or bots or logs in the same format as your favorite IRC client.

17.2 Substitutes

The configuration values define the syntax that is used to write out log file lines. In these configuration values, values beginning with a '%' can be substituted.

The following characters are allowed after a percent sign for all types of lines:

h Current time of day, hours field.

M Current time of day, number of minutes.

s Current time of day, number of seconds.

n Nick originating the line (saying the message, doing the kick, quitting, joining, etc).

u Hostmask of the user originating the line.

N Name of the current IRC network.

S Name of the server (as set by the transport).

% Percent sign

0,1,2,3,4,5,6,7,8,9 Substituted with the respective argument in the IRC line.

@ Replaced by channel name if the message is directed to a channel, the nick name to which the message is being sent, or the name of the sender of the message when the receiver is the user running ctrlproxy.

This substitute will be the name of the first channel on which the user is active if the line type is NICK or QUIT.

Each type of line also has some variables of it's own that it substitutes.

17.2.1 join

%c Name of the channel the user joins.

17.2.2 part

%c Name of the channel the user is leaving.

%m Comment

17.2.3 kick

%t Nick of the user that is being kicked.

%c Channel the user is being kicked from.

%r Reason the user is being kicked.

17.2.4 quit

%m Comment.

17.2.5 topic/notopic

%c Name of the channel of which the topic is being changed.

%t The new topic. Only set for 'topic', not for 'notopic'.

17.2.6 mode

%c Name of user or channel of which the mode is being changed.

%p Change in the mode, e.g. `+oie`

%t Target of which the mode is being changed.

To retrieve any additional arguments for a MODE command, use *%1*, *%2*, etc.

17.2.7 notice/privmsg/action

%t Name of channel or nickname of user to which the notice/privmsg/ or action is being sent.

%m Message that is being sent.

17.2.8 nick

%r New nickname the user is changing his/her name to.

17.3 Configuration

The following XML elements are supported:

logfile Path to the logfile that will be written. Supports substitution depending on the type of line that is being parsed.

join Format to use for lines where a user joins a channel.

part Format to use for lines where a user leaves a channel.

msg Format to use for 'regular' messages - when a user says something.

notice Format to use for notices.

action Format to use for CTCP actions (e.g. /me ...)

mode Format to use for MODE changes (including bans)

quit Format to use for quit lines.

kick Format to use for kicks.

topic Format to use for topic changes to a valid topic

notopic Format to use when the topic is unset.

nickchange Format to use when a user changes his/her nick name.

17.4 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="log_custom">
         <logfile>/home/jelmer/log/ctrlproxy/%@</logfile>
         <join>%h%M%s -!- User %n [%u] has joined %c</join>
         <part>%h%M%s -!- User %n [%u] has left %c [%m]</part>
         <quit>%h%M%s -!- User %n [%u] has quit [%m]</quit>
         <action>%h%M%s * %n %m</action>
      </plugin>
      <plugin autoload="1" file="socket"/>
   </plugins>

   <networks>
      <network name="OFTC">
         <servers><ipv4 host="irc.oftc.net"/></servers>
         <channel name="#flood"/>
      </network>
   </networks>
</ctrlproxy>
```

Chapter 18

chapter

NickServ

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

18.1 Description

This module takes care of registration with NickServ and ghosting older connections.

18.2 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="nickserv"/>
      <plugin autoload="1" file="socket"/>
   </plugins>

   <networks>
      <network name="OFTC">
         <servers><ipv4 host="irc.oftc.net"/></servers>
         <channel name="#flood.nl" autojoin="1"/>
         <nickserv>
            <nick name="foo" password="secret"/>
         </nickserv>
      </network>
   </networks>
</ctrlproxy>
```

Chapter 19

chapter

Flood protection module

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

19.1 Description

This module makes sure at most 1 message is sent to the server in a certain period of time.
A child element of a server element named "queue_speed" contains the number of milliseconds the client has to wait before sending a new message. [networks]

[/networks]

19.2 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="antiflood"/>
      <plugin autoload="1" file="socket"/>
   </plugins>

   <networks>
      <network name="OFTC">
         <queuespeed>2200</queuespeed>
         <servers><ipv4 host="irc.oftc.net"/></servers>
         <channel name="#flood"/>
      </network>
   </networks>
</ctrlproxy>
```

Chapter 20

chapter

Time reporter

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

20.1 Description

Very simple module that prints the time before every message. Such messages are very useful for backlogs. That way, you can know at approximately which time something was said.

20.2 Configuration

The following XML elements are supported:

format Format of the timestamp. See the manual page of the strftime() function for what is supported.

20.3 Example configuration

```
<ctrlproxy>
   <plugins>
      <plugin autoload="1" file="report_time">
         <format>%h:%m:%s</format>
         <interval>300</interval>
      </plugin>
   </plugins>
</ctrlproxy>
```

Chapter 21

chapter

Reading MOTD from file

Version: 0.1

Author: Jelmer Vernooij <mailto:jelmer@vernstok.nl>

Homepage: <http://ctrlproxy.vernstok.nl/>

21.1 Description

This module reads a MOTD from a specified location and shows that to the user when (s)he logs into ctrlproxy.

21.2 Configuration

The following XML elements are supported:

file File to read the MOTD from. By default, sharedir/motd is used. (sharedir is usually /usr/share/ctrlproxy

Chapter 22

chapter

Python scripting support

Version: 0.5

Author: Daniel Poelzleithner <mailto:ctrlproxy@poelzi.org>

Homepage: <http://ctrlproxy.vernstok.nl/>

Requirements: python

22.1 Description

This module provides python scripting support inside ctrlproxy.

22.2 Configuration

The following XML elements are supported:

redirect-stdout Redirects the standard output in python to the log of ctrlproxy

redirect-stderr Redirects the error output in python to the log of ctrlproxy

22.3 Function callbacks

- Detecting client changes (You connect/disconnect to ctrlproxy,..)
- Server connection (ctrlproxy connects/disconnects to a server)
- Motd (not functional currently :( ) the scripts sends some motd to the client
- Admin commands (the script provides admin commands)

This means, the registered function is called when a change occures.
Example:

```
def myNewClient:
   print "Year, a new client"

ctrlproxy.add_new_client_hook(myNewClient)
```

Everytime a client connects to ctrlproxy, this function is called :)
For working on the IRC data, you have two interfaces:
The Low Level Interface is, to register a rcv function with add_rcv_hook(myFunction)
The High Level, and much faster Interface is to overwrite the ctrlproxy.Event Object.

```
class myEvent(ctrlproxy.Event):
   def onChannelJoin(self, line, data):
      print "yeah, i got a join :)\n"
      print data
   def onElse(self, line, data):
      print "something else\n"
      print data
   def onRcv(self, line, data):
      print "R: %s" %data
   #  print "got something"
   def onWho(self, line, data):
      print "who"
      print data
   def onWhois(self, line, data):
      print "whois ?"
      print data

n = myEvent()
ctrlproxy.add_event_object(n);
```

Every method of the Object gets called when the specific Event occures. For example:
You type "/whois poelzi_" in your irc client. The example myEvent will print following message:

```
** INFO: whois ?
** INFO: [['311', 'poelzi', 'poelzi_', 'poelzi',
'cruor.intra.poelzi.org', '*', 'poelzi'], ['319', 'poelzi', 'poelzi_',
'@#bla @#test '], ['312', 'poelzi', 'poelzi_', 'irc.localnet',
'http://www.debian.org/'], ['317', 'poelzi', 'poelzi_', '10393',
'1072210235', 'seconds idle, signon time'],['318', 'poelzi', 'poelzi_',
'End of /WHOIS list.']]
```

Yes, it looks weird, but it isn't: One whois replay is more then one line, the Event object collect them for you and call the function with a List Object that contains every line (these are the outer brackets) Because every line is preparsed, its a List to.

```
['311', 'poelzi', 'poelzi_', 'poelzi', 'cruor.intra.poelzi.org', '*',
'poelzi'], ['319', 'poelzi', 'poelzi_', '@#bla @#test ']

['312', 'poelzi', 'poelzi_', 'irc.localnet', 'http://www.debian.org/']

['317', 'poelzi', 'poelzi_', '10393', '1072210235', 'seconds idle,
signon time']]

['318', 'poelzi', 'poelzi_', 'End of /WHOIS list.']
```

Every message except the onRcv which is always only one line, is a List in a List, even, when there is only one line in the result.

22.4 Where the data comes from....

The Python interface is a realtime interface to the data structures in ctrlproxy. This means, when you access some data in ctrlproxy, thats the current state. On the other hand, you can not save these object. You can make a copy of the data in normal python objects when you replicate them, but not the ctrlproxy object itself. this means, you can access the objects ctrlproxy passes as long you stay in the current function, when the function return, the object could be invalid (mostly will).

But this shouldn't bother you, because you can request everything.

ctrlproxy.list_networks() returns a list of network names.

ctrlproxy.get_network("name") returns the network object. This is the head of everything and you can walk down the tree.

```
n = ctrlproxy.get_network("test")

print "XMLconf:",n.xmlConf
print "Mymodes:",n.mymodes
print "Servers:",n.servers
print "Hostmask:",n.hostmask
print "Channels:",n.channels
print "Authenticated:",n.authenticated
print "Clients:",n.clients
print "Current_server:",n.current_server
print "Listen:",n.listen
print "Supported_modes:",n.supported_modes
print "Features:", n.features
```

You see that you can access the network really easy.

```
n.disconnect()
```

and your network is offline :)

```
n.clients[0].send_notice("i like ctrlproxy really much")
```

the first client connected to network test becomes this notice.

```
n.clients[0].send_notice("good bye !!")
n.clients[0].disconnect()
```

but now he is gone.
you can send directly into the network.

```
n.send("whois test test",ctrlproxy.TO_SERVER)
```

or something to all clients connected to this network

```
n.send("PRIVMSG test hahahahah my message",ctrlproxy.TO_CLIENTS)
```

the xml here, too:

The xml nodes from ctrlproxy become xml Objects in python. These Objects have direct read/write access to the xmlNodes in ctrlproxy. This means, you can reconfigure ctrlproxy throu them. Look at the libxml2-python documentation.

Readonly:

Only the Line object has write access, all other changes are done through function. This mean you can't write:

```
"del network.client[0]"
```

to kill a client for example.

the only exception is the line object. you can do:

```
line.options = line.option | ctrlproxy.LINE_DONT_SEND
```

the line will not be send to the target.

22.5 Threads

In general, you can use threads. There is a know bug i wasn't able to solve yet. When a thread dies, the process is still in process list as defunct with 0 memory usage.

In general you should avoid threads as much as possible, or look that the theads to not die :)

A example for thread usage:

You write a dcc plugin that save files which are sent to you on the server. To recive the data, create one thread for all incomming data, that never dies, and append to him the jobs he has to do.

As soon as i know how to recieve the exit code of the thread, this is solved :)

22.6 Something general

The stdout and stderr streams from python are wrapped to the general logging functions of ctrlproxy. These created the "** INFO: " at the message. When you load the noticelog plugin, these messages are pipped to the client. So, you should not print every line you get via a rcv method, or you will create a endless loop :)

22.7 Requirements

- python2.3 (maybe 2.2 works too)
- python2.3-dev (python headers)
- libxml2-python

22.8 Example configuration

```
<ctrlproxy>
   <plugin autoload="1" file="libpython">
      <load>
         <file>/home/poelzi/Projects/ctrlproxy/example/test.py</file>
         <argument name="test">blubbdada</argument>
      </load>
```

```
    </plugin>
  </ctrlproxy>
```

22.9 Configuration

Each argument and value pair becomes a entry in the ctrlproxy.args array. This array is runtime and script specific.

When you load a python script via the admin command.

```
/ctrlproxy python load /home/bla/.ctrlproxy/test.py abc="huhu dada"
hihi=" blabla "
```

will load test.py with the argument dict: {"abc":"huhu dada","hihi":" blabla"}

22.10 Planned features

- del_____hook
- Documentation
- implement some more functions to ctrlproxy

Part IV

Writing your own modules

Chapter 23

General

As has been said in the introduction, ctrlproxy is easily extendible. At the time of writing, there are nine modules available.

The simplest possible module would be:

```
#include <ctrlproxy.h>

gboolean init_plugin(struct plugin *p)
{
   /* Do something */
   return TRUE;
}

gboolean fini_plugin(struct plugin *p)
{
   /* Free my structures here */
   return TRUE;
}
```

The init_plugin function is called when the module is loaded. In this function, you should register whatever functions the module provides, such as a 'message handler' or a linestack backend. You can use the *data* member of the plugin struct to store data for your plugin. This function should return a boolean: false when initialisation failed or true when it succeeded.

THe fini_plugin function is called before the module is unloaded. In this function, you should free the data structures your module is using and make sure there are no other pointers in ctrlproxy pointing to functions or data structures from your module. For example, unregister transports or hooks.

The fini_plugin should return a boolean as well. This value should be true if the unloading may preceed, or false if there are reasons ctrlproxy should not attempt to unload the module (such as resources that are currently in use, etc).

23.1 Building and installing

A module is in fact a shared library that's loaded at run-time, when the program is already running. The `.so` file can compiled with a command like:

```
$ gcc -shared -o foo.so input1.c input2.o input3.c
```

23.2 Message handler functions

A message handling function is a function that is called whenever ctrlproxy receives an IRC message. The only argument this function should have would be a line struct.

Flags can be set on the line (the field in the struct to use is called 'options') to influence the handling of the packet by the rest of ctrlproxy. At the time of writing, the following two flags are available:

LINE_DONT_SEND Continue processing, but do not send this line.

LINE_STOP_PROCESSING Immediately stop processing the line (passing it to other message handlers). Implemented as of version 2.5.

LINE_NO_LOGGING Modules that do logging should ignore this line. This may be used for PRIVMSG's that are not interesting for logs, such as timestamps that are being printed.

There is one other option that can be specified, but is only useful when sending your own messages:

LINE_IS_PRIVATE Do not send this line to other clients currently connected.

23.2.1 Registering a message handler

All IRC lines that ctrlproxy receives and sends are passed thru so-called 'filter functions'. These functions can do things based on the contents of these lines, change the lines or stop further processing of these lines.

To add a filter function, call 'add_filter'. To remove the filter function again (usually when your plugin is being unloaded) call 'del_filter'.

Example:

```
...
add_filter("my_module", my_message_handler);
...
```

The prototype for the message handling function in the example above would look something like this:

```
static gboolean my_message_handler(struct line *l);
```

Your message handler should return TRUE if the rest of the filter functions should also see the message and FALSE if ctrlproxy should stop running filter functions on the given line struct.

NOTE

These hooks are executed *before* the data as returned by find_channel() and find_nick() is updated

23.2.2 Registering a new client/server or lose client/server handler

A module can also register a function that should be called when a new client connects or when a client disconnects and when the server has successfully connected to the client or when the connection to the client is broken.

```
typedef gboolean (*new_client_hook) (struct client *);
typedef void (*lose_client_hook) (struct client *);
void add_new_client_hook(char *name, new_client_hook h);
void del_new_client_hook(char *name);
void add_lose_client_hook(char *name, lose_client_hook h);
void del_lose_client_hook(char *name);

typedef void (*server_connected_hook) (struct network *);
typedef void (*server_disconnected_hook) (struct network *);
void add_server_connected_hook(char *name, server_connected_hook);
void del_server_connected_hook(char *name);
void add_server_disconnected_hook(char *name, server_disconnected_hook);
void del_server_disconnected_hook(char *name);
```

The prototypes of these functions pretty much speak for themselves. If a new_client_hook function returns FALSE, the client will be denied access.

23.2.3 Registering a initialization function

The initialization hooks are called after ctrlproxy has been initialized - all plugins are loaded, all networks have been loaded.

```
typedef void (*initialized_hook) (void);
void add_initialized_hook(initialized_hook);
```

23.2.4 Registering a MOTD function

MOTD functions are functions that add one or more lines to the MOTD that is sent to a client.

A module can register a MOTD function using the add_motd_hook() and del_motd_hook() functions, that work similar to the add_new_client_hook() and del_new_client_hook() functions documented above.

A motd function should return a dynamically allocated array containing dynamically allocated nul-terminated strings that should be added to the MOTD.

23.3 Log functionality

Ctrlproxy uses GLib's logging functions. Read the related section in the GLib documentation for details.

23.4 Storing data

Paths to data should be configurable, but default to the file/directory name returned by `ctrlproxy_path()`. The argument to this function should be the name of the subsystem.

All top level directories have been created when this function returns.

If NULL was returned, one or more directories could not be created.

23.5 Debugging

Two very useful utilities are **valgrind** and **gdb**.

If you're running from gdb, make sure you have set the following:

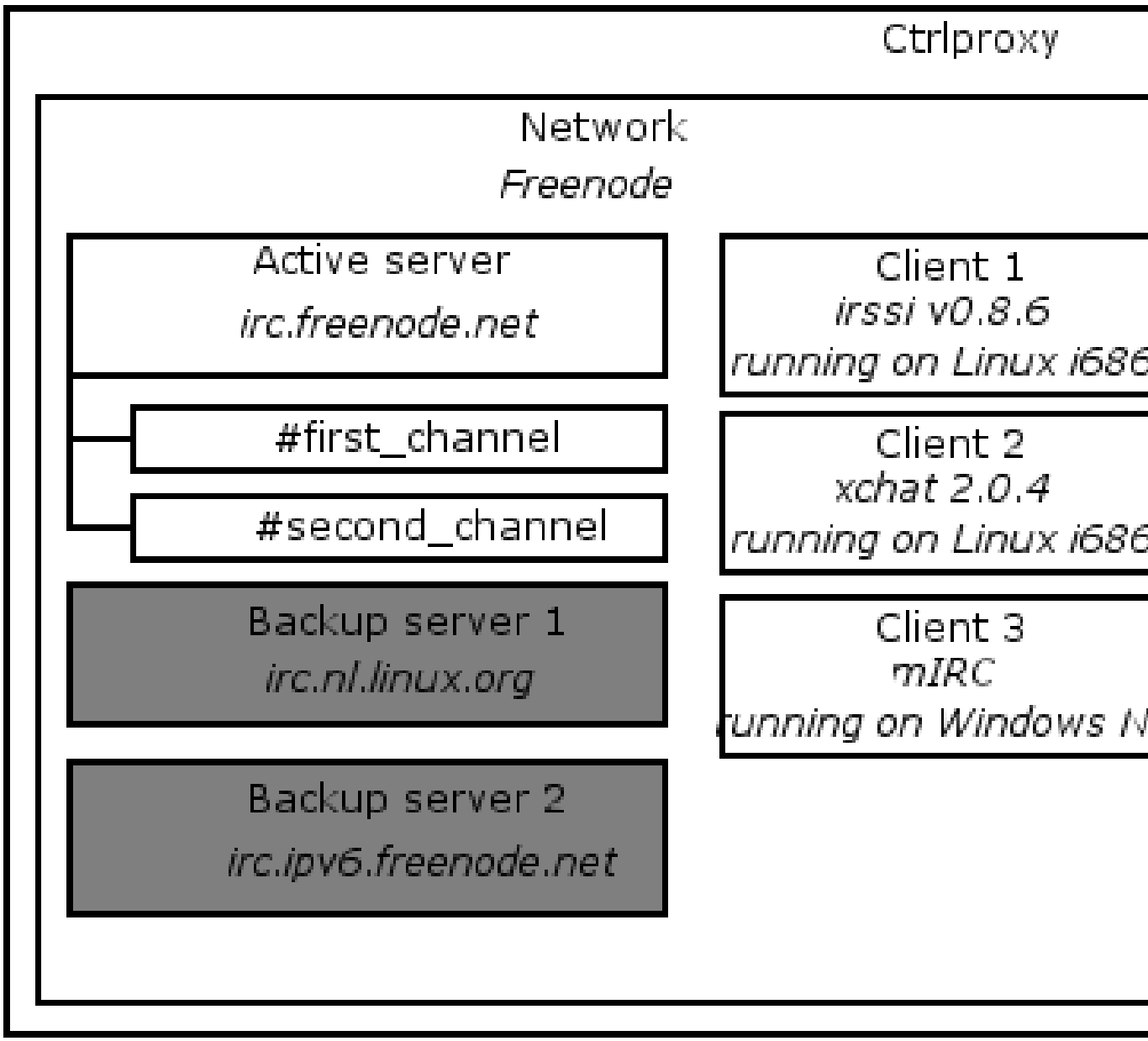
```
handle SIGPIPE nostop
handle SIGINT nostop
```

Chapter 24

API

This chapter describes the functions that are available for third-party plugin writers.

24.1 Main structs



24.1.1 struct client

```
struct client {
   struct network *network;
   char authenticated;
   struct transport_context *incoming;
   time_t connect_time;
};
```

Describes one single client connection to ctrlproxy.

struct network *network Pointer to network struct this client belongs to.

char authenticated Indicates whether the client has been authenticated by ctrlproxy. (By sending the correct "PASS ..." line). If set to 0, the client has not been authenticated, if set to 1, the client has been successfully authenticated. A value of 2 means the client has disconnected.

struct transport_context *incoming Transport context to be used to communicate with the client.

time_t connect_time Contains unix timestamp of the moment the client did its initial connect. This field is used to kick clients that have not authenticated after one minute.

24.1.2 struct network

```
struct network {
   xmlNodePtr xmlConf;
   char modes[255];
   xmlNodePtr servers;
   char *hostmask;
   GList *channels;
   char authenticated;
   GList *clients;
   xmlNodePtr current_server;
   xmlNodePtr listen;
   char *supported_modes[2];
   char **features;
   struct transport_context *outgoing;
   struct transport_context **incoming;
};
```

Describes an IRC network that ctrlproxy is connected to.

xmlNodePtr xmlConf; Points to XML node with configuration for this network.

char modes[255]; Array with modes of the user on this network. For modes that have been set, the index in this array has been set to 1. The rest of the array is set to 0.

For example, if mode "i"(invisible) is set on this user, "modes['i']" is set to 1.

xmlNodePtr servers; Pointer to XML node <servers> for this server.

char *hostmask; Hostmask that ctrlproxy uses to communicate to the server.

GList *channels; List of "struct channel" pointers with channels the user has joined on this network.

char authenticated; Indicates whether the connection to this network is established. It is set to true after a 004 message has been received.

GList *clients; List of "struct client" pointers with all the clients that have connected to ctrlproxy for this network.

xmlNodePtr current_server; Pointer to XML node that contains the configuration data of the current server ctrlproxy is connected to for this network.

xmlNodePtr listen; Pointer to XML node <listen>.

char *supported_modes[2]; Contains 2 arrays of modes that is supported by the remote server. This list is sent by the server after the connection has just been set-up.

char **features; Array of options supported by the server. Same format as unix environment variables, though a value is not required.

struct transport_context *outgoing; Transport context to use to communicate with the remote server.

struct transport_context **incoming; List with transport contexts for the clients that are currently connected to ctrlproxy for this server.

24.2 State data

This section covers everything related to the current (known) state information of the network the user is on.

24.2.1 struct nick

```
struct nick {
   char *name;
   char mode;
};
```

Covers one nick in a certain channel. Mode is either a space, indicating the user has no special rights, a '@' if the user is an operator or a '+' if the user has voice.

24.2.2 struct channel

```
struct channel {
   xmlNodePtr xmlConf;
   char *topic;
   char mode;
```

```
    char *modes[255];
    char introduced;
    long limit;
    char *key;
    GList *nicks;
};
```

Covers one channel at a certain network that the user is currently on. Here is a small list with explanation of the various fields.

xmlNodePtr xmlConf Pointer to XML node describing this channel.

char mode Indicates whether the channel is private or secret.

char *topic Pointer to string containing the topic of this channel. NULL if no topic has been set or if the topic is unknown.

char *modes[255] Modes that have been set on this channel. FIXME

char introduced Reserved for use by replication functions. Private. Do not use.

long limit Maximum number of users on the channel. 0 means no limit has been set.

char *key Key users have to enter to enter the channel. If no key is required, this field is set to NULL.

GList *nicks List of "struct nick", one for each user that is joined to the channel.

24.2.3 find_channel()

```
struct channel *find_channel(struct network *st, char *name);
```

Returns a pointer to the struct of the channel with the specified name on the specified network. Returns NULL if no channel struct was found.

Note that this function only works for channels the user has currently used.

24.2.4 find_nick()

```
struct nick *find_nick(struct channel *c, char *name);
```

Find data pointer to "struct nick" of the user with the specified name on the specified channel.
If the user was not found, NULL is returned.

24.2.5 gen_replication_network()

```
GSList *gen_replication_network(struct network *s);
```

Generates double-linked list of strings that need to be send to a client to give it a good view of the channels that have been joined, the users on those channels and the modes of those channels.

24.2.6 gen_replication_channel()

```
GSList *gen_replication_channel(struct channel *s, char *hostmask, char *nick);
```

Generates double-linked list of strings that need to be send to a client to give it a good view of a channel that ctrlproxy has joined. hostmask should be the name of the server the messages are coming from and nick the name of the user the replication is sent to.

24.2.7 default_replicate_function

```
void default_replicate_function (struct network *, struct transport_context *);
extern void (*replicate_function) (struct network *, struct transport_context *);
```

Default replication function. What this basically does is sending the strings returned by gen_replication() to the specified transport_context.

24.3 Maintaining the main process

```
extern GList *networks;
extern xmlNodePtr xmlNode_networks, xmlNode_plugins;
extern GList *plugins;
extern xmlDocPtr configuration;
extern GHookList data_hook;
```

Pointers to various useful varables. The *xmlNode_** variables point to the <networks> and <plugins> elements in the rc file.

configuration points to the top level XML document.

data_hook can be used to register a function that should be called whenever ctrlproxy receives or sends IRC messages.

plugins and *networks* contain lists to all "struct plugin"s and "struct network"s, respectively.

24.3.1 network_add_listen()

```
void network_add_listen(struct network *, xmlNodePtr);
```

Add listener to specified network with configuration specified in xmlNodePtr.

24.3.2 save_configuration()

```
void save_configuration();
```

Save the current state of the XML configuration of ctrlproxy to the same file it was loaded from.

24.3.3 load_plugin()

```
gboolean load_plugin(xmlNodePtr);
```

Load plugin with specified configuration. xmlNodePtr should point to a <plugin> element.

24.3.4 unload_plugin()

```
gboolean unload_plugin(struct plugin *);
```

Try to unload the specified plugin. Not all plugins support this at the moment. Returns TRUE if the unloading succeeded, FALSE otherwise. Failure of unloading may be caused by resources that are still in use.

24.3.5 plugin_loaded()

```
gboolean plugin_loaded(char *name);
```

Returns TRUE if a plugin with the specified name was loaded.

24.4 Transports

```
struct transport;
struct transport_context;
```

24.4.1 register_transport()

```
void register_transport(struct transport *);
```

Register the specified transport. See the next chapter for details.

24.4.2 transport_connect()

```
struct transport_context *transport_connect(const char *name, xmlNodePtr p, receive_handle
```

Connect using the transport with name *name* and configuration *p*.
The receive_handler and disconnect_handler will be called when new data is received and when the remote has disconnected, respectively. The *data* pointer will be passed to the disconnect and receive handlers.

24.4.3 transport_listen()

```
struct transport_context *transport_listen(const char *name, xmlNodePtr p, newclient_handl
```

Listen for incoming connections using the transport with name *name*, which has configuration *p*.
The newclient_handler will be called whenever a new client connects to the transport. *data* will be passed to it.

24.4.4 transport_free

```
void transport_free(struct transport_context *);
```

Disconnect the specified transport and free all data associated with it.

24.4.5 transport_write()

```
int transport_write(struct transport_context *, char *l);
```

Write specified line to the transport. *l* has to be null-terminated!

24.4.6 transport_set_disconnect_handler()

```
void transport_set_disconnect_handler(struct transport_context *, disconnect_handler),
typedef void (*disconnect_handler) (struct transport_context *, void *data);
```

Set function to call when the remote closes the transport.

24.4.7 transport_set_receive_handler()

```
void transport_set_receive_handler(struct transport_context *, receive_handler);
typedef void (*receive_handler) (struct transport_context *, char *l, void *data);
```

Set function to call when new data is received on the socket. *l* will be a null-terminated string.

24.4.8 transport_set_newclient_handler()

```
typedef void (*newclient_handler) (struct transport_context *, struct transport_contex
void transport_set_newclient_handler(struct transport_context *, newclient_handler);
```

Set function to call whenever a new client connects to the specified (listening) transport context.

24.4.9 transport_set_data()

```
void transport_set_data(struct transport_context *, void *);
```

Set user data to pass to the various callback functions (receive_handler, disconnect_handler, newclient_handler).

24.5 Line parsing/creation/handling

These functions all have to do with manipulating line structs. Pretty much all internal functions of ctrlproxy work with these instead of manipulating plain strings.

```
struct line {
   enum data_direction direction;
   int options;
   struct network *network;
   struct client *client;
   const char *origin;
   char **args; /* NULL terminated */
   size_t argc;
};
```

```
/* for the options fields */
#define LINE_IS_PRIVATE      1
#define LINE_DONT_SEND       2
#define LINE_STOP_PROCESSING 4

enum data_direction { UNKNOWN = 0, TO_SERVER = 1, FROM_SERVER = 2 };
```

enum data_direction direction; Direction of this line. A value of TO_SERVER means it's going to the server, FROM_SERVER means it's coming from a remote IRC server. UNKNOWN is used in cases where the direction is not known.

int options; Sum of one of LINE_IS_PRIVATE, LINE_DONT_SEND and LINE_STOP_PROCESSING. LINE_IS_PRIVATE means this line was send by a client and should not be sent to the other clients. LINE_DONT_SEND should be used to tell ctrlproxy to not send this line to its destination (either client or server). LINE_STOP_PROCESSING will stop further filtering of the line.

struct network *network; Points to the network this line came from or is going to.

struct client *client; Points to the client this line came from, if any. Set to NULL if unknown.

const char *origin; Hostmask of the user who sent the message. NULL if unknown.

char **args; IRC arguments/commands in an array. Last element is set to NULL.

size_t argc; Contains number of arguments/commands in *args*.

24.5.1 linedup()

```
struct line *linedup(struct line *l);
```

Duplicate the given line struct.

24.5.2 irc_parse_line()

```
struct line * irc_parse_line(char *data);
```

Takes a string as sent by an IRC client or an IRC server and generates a struct line.

24.5.3 virc_parse_line()

```
struct line * virc_parse_line(char *origin, va_list ap);
struct line *irc_parse_line_args( char *origin, ... );
gboolean irc_send_args(struct transport_context *, ...);
```

Generates a line struct with the hostmask specified in *origin* or NULL if none should be set.

For virc_parse_line(), the *ap* should be a list of strings that are each that are a seperate part of the IRC line. The last argument should be NULL to indicate the end of the list.

irc_parse_line_args() is similar to virc_parse_line(), except that now the commands don't need to be passed in a *va_list*, but can be passed as arguments.

irc_send_args() sends the specified commands, terminated by a NULL to the specified transport_context.

24.5.4 irc_line_string()

```
char *irc_line_string(struct line *l);
char *irc_line_string_nl(struct line *l);
```

Generate a string representation of a line struct in the format used by IRC clients and servers.

irc_Line_string_nl() is similar to irc_line_string(), except that it adds a newline and a carriage-return to the string (\r\n).

24.5.5 line_get_nick()

```
char *line_get_nick(struct line *l);
```

Get the nick name of the user that sent *l* or NULL if the nick name was unknown.

24.5.6 free_line()

```
void free_line(struct line *l);
```

Free all data associated with *l*.

24.5.7 irc_sendf()

```
gboolean irc_sendf(struct transport_context *, char *fmt, ...);
struct line *irc_parse_linef( char *fmt, ... );
```

irc_sendf() sends the specified transport_context a IRC line. fmt is a printf-like string and the remaining arguments correspond to the data in *fmt*. See the printf manpage for details.

irc_parse_linef() is similar, but instead of sending the string it generates a struct line and returns it.

24.5.8 irc_send_line()

```
int irc_send_line(struct transport_context *, struct line *l);
```

Send the specified line to the specified transport_context.

24.5.9 clients_send()

```
void clients_send(struct network *, struct line *, struct transport_context *exceptio
```

Send the specified line to all clients on the specified network, except for the client with transport_context *exception*. *exception* can be NULL.

24.6 General purpose functions

24.6.1 list_make_string()

```
char *list_make_string(char **l);
```

Creates a string with all the elements in string array `l`, seperated by spaces. The last element in `l` should be NULL.

24.6.2 xmLFindChildByName()

```
xmlNodePtr xmlFindChildByName(xmlNodePtr parent, const xmlChar *name);
```

Find a child node of the XML node `parent` that is an element with name `name` and return the xml Node pointer of it.

Returns NULL if no such child was found.

Chapter 25

Transports

Transports are ctrlproxy's own layer for sending and receiving data in a way that is independant of the implementation underneath (IP, UNIX sockets, etc). Since transports are aimed at IRC-only data, they work with lines (*char **) and not with lengths, etc. Data is only passed to the main process when a complete line is in, not parts of it.

Implementors of a certain transport backend should call **register_transport()** with a pointer to a *struct transport_ops*.

25.1 Transport contexts

The following struct is passed to all transport functions.

```
struct transport_context {
   struct transport_ops *functions;
   xmlNodePtr configuration;
   void *data;
   void *caller_data;
   disconnect_handler on_disconnect;
   receive_handler on_receive;
   newclient_handler on_new_client;
};
```

The *configuration* xmlNodePtr contains configuration for this specific instance of the transport. The *data* pointer can be used by the transport to store instance-specific data. The three 'handler' functions should be called whenever one of these events occur. Please note that you have to check for available data yourself. See the documentation about the main context in GLib for details on registering polling and idle functions.

25.2 Functions to provide

A transport struct should contain function pointers to the following functions:

connect This function should connect to a IRC server.

listen This function should make the transport waiting for incoming connections.

write Function to write/send the specified line using the transport.

close Close (if necessary) any outstanding ports, file handles, etc. This function is always called before a transport is freed.

Each of the function pointers listed above can be set to NULL, to indicate that the function is not implemented.

25.3 Callbacks to call

The following callbacks, which are listed in transport_context should be called by your transport. The "data" argument in all of these calls should be the "callerdata" member field of the struct transport_context.

typedef void (*disconnect_handler) (struct transport_context *, void *data); Called when the remote host closes the connection.

typedef void (*receive_handler) (struct transport_context *, char *l, void *data); Called when a new line with contents "l" has arrived.

typedef void (*newclient_handler) (struct transport_context *, struct transport_context *, void *data); Function to be called when a new client has connected to the transport. The second argument contains a pointer to a new transport_context which can be used to talk to the new client.

Chapter 26

Line Stacks (lists of lines)

Line stacks are structures that contain a bunch of line structs. They are used for various things inside ctrlproxy, but usually for replication.

You can do a few things with line stacks: add lines to them, retrieve all lines, send all lines to a *transport_context* and clear the stack.

linestack has various implementations (backends), each with their own advantages and disadvantages.

26.1 General functions

26.1.1 linestack_new

```
struct linestack_context *linestack_new(char *name, char *args);
```

Create a new linestack_context based on the backend with the specified name and the specified arguments. If name is NULL, the first avallable backend will be used.

26.1.2 linestack_get_linked_list

```
GSList *linestack_get_linked_list(struct linestack_context *);
```

Retrieves a single linked listed with all lines in the specified linestack_context.

26.1.3 linestack_send

```
void linestack_send(struct linestack_context *, struct transport_context *);
```

Send all lines in the specified linestack_contact using the specified transport_context.

26.1.4 linestack_destroy

```
gboolean linestack_destroy(struct linestack_context *);
```

Free all memory associated with the specified linestack_context and/or close any resources the linestack_context is using.

26.1.5 linestack_clear

```
gboolean linestack_clear(struct linestack_context *);
```

Remove all lines from the specified linestack_context.

26.1.6 linstack_add_line

```
gboolean linestack_add_line(struct linestack_context *, struct line *);
```

Add the specified line struct to the specified linestack_context.

26.1.7 linestack_add_line_list

```
gboolean linestack_add_line_list(struct linestack_context *, GSList *);
```

Add all the line structs in the specified linked list to the specified linestack_context.

26.2 Writing your own linestack backend

```
/* linestack.c */
struct linestack_context;
struct linestack_ops {
   char *name;
   gboolean (*init) (struct linestack_context *, char *args);
   gboolean (*clear) (struct linestack_context *);
   gboolean (*add_line) (struct linestack_context *, struct line *);
   GSList *(*get_linked_list) (struct linestack_context *);
   void (*send) (struct linestack_context *, struct transport_context *);
   gboolean (*destroy) (struct linestack_context *);
   };

struct linestack_context {
   struct linestack_ops *functions;
   void *data;
};
```

26.2.1 linestack functions a backend should provide

To add a linestack backend, a module should fill in a linestack struct and register that. A linestack struct contains the following fields:

name Name of the backend. Should _NOT_ be NULL.

init Should point to a function that will be called after a transport_context of this type has been created. The optional argument will contain some configuration details specified by the user (for example, a file name for a file-based linestack backend). The argument may be NULL.

clear Should point to a function that will remove all lines from the specified linestack_context.

add_line Should point to a function that will add the specified line struct to the specified linestack_context.

get_linked_list Should point to a function that will return a single linked list with all the current lines in the specified linestack_context.

send Should point to a function that will sned all the lines in the specified linestack_context to the specified transport_context. If set to NULL, a dummy function will be used that sends all lines returned by get_linked_list() to the transport_context.

destroy Should point to a function that frees all resources that are in used by the specified linestack_context. May be NULL, if no resources need to be freed.

26.2.2 register_linestack

```
void register_linestack(struct linestack_ops *);
```

Register a new linestack backend (ie, make it available, not necessarily use it), described by the specified linestack struct.

26.2.3 unregister_linestack

```
void unregister_linestack(struct linestack_ops *);
```

Register a new linestack backend (ie, make it available, not necessarily use it), described by the specified linestack struct.