The ControlProxy Manual

by Jelmer Vernooij

The ControlProxy Manual

by Jelmer Vernooij

Published $Date: 2003/10/05 02:11:08 $

This documentation is a draft. It is full of typo’s and grammar errors. It might be unclear at some points.

Future versions of ctrlproxy will include an improved version of this document.

Meanwhile, all comments, questions and updates are welcome at jelmer@vernstok.nl [mailto:jelmer@vernstok.nl].

Table of Contents

Chapter 1. Introduction

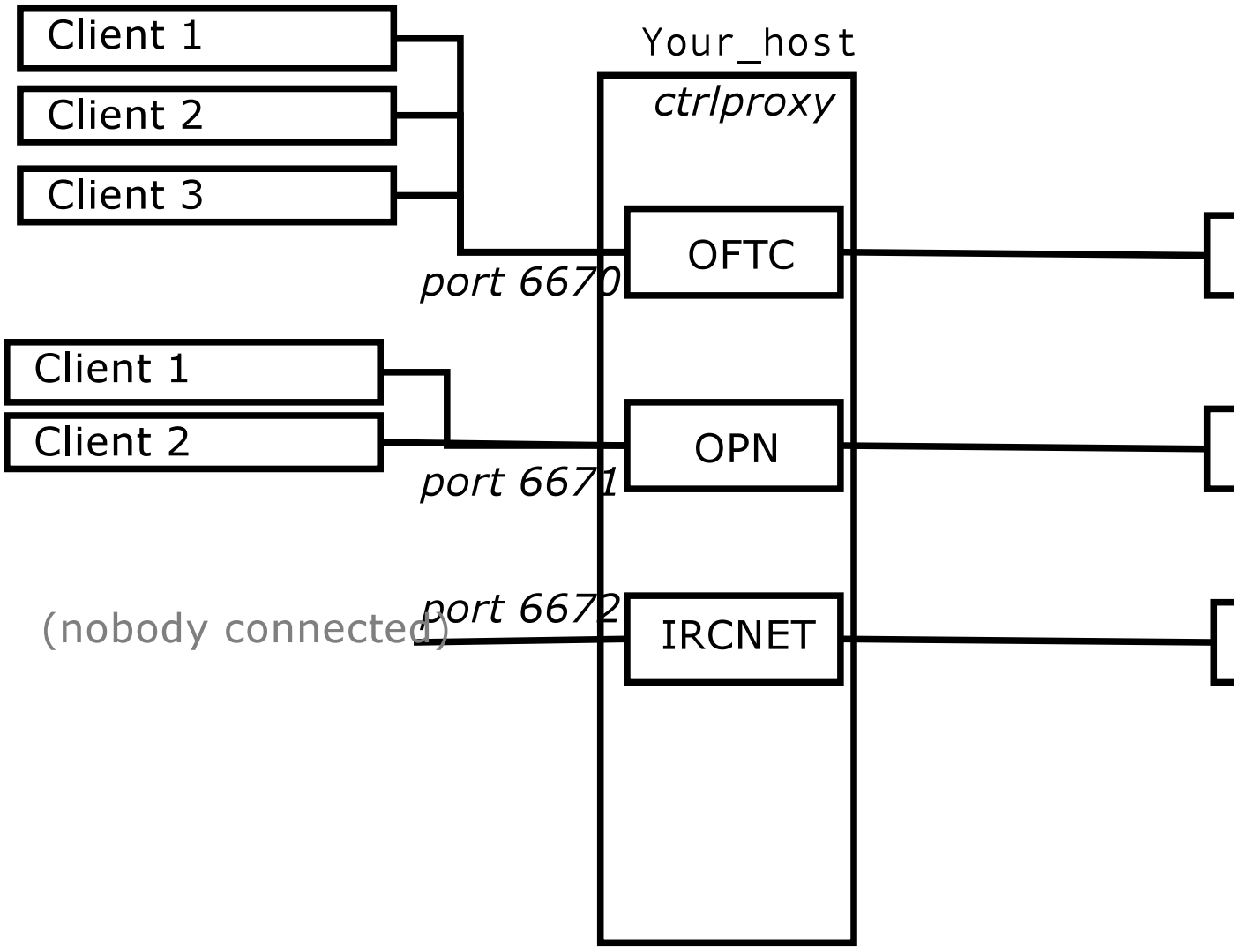
Why ctrlproxy?

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.

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.



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 support

Requirements

- libpopt
- GNU glib
- libxml2

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

Part I. Installation

Table of Contents

Chapter 2. Installation

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.

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

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-2.5.tar.gz
ctrlproxy-2.5/AUTHORS
...
```

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

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!

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

Table of Contents

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.

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.

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.

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>
```

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>
```

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>
```

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>
```

Part III. Modules

Table of Contents

Chapter 4. admin module

Remote administration

Version: 0.1

Author: Jelmer Vernooij [mailto:]

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

Description

This module provides a simple interface for remote administration of ControlProxy. Commands can be executed by 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.

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.

DIE Disconnect all clients and servers and exit ctrlproxy.

DISCONNECT <network> Disconnect from the specified network.

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

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.

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
```

Chapter 5. auto-away module

Automagic away

Version: 0.1

Author: Jelmer Vernooij [mailto:]

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

Description

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

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.

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. Please
leave me a message. Thanks!</message>
 </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 6. ctcp module

Standard CTCP module

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://jelmer.vernstok.nl/ctrlproxy/

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 7. log_irssi module

Irssi-style log files

Version: 0.1

Author: Jelmer Vernooij [mailto:]

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

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.

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.

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 8. repl_memory module

Replication from memory

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://jelmer.vernstok.nl/ctrlproxy/

Description

Sends all messages to a new client that were sent since the last time the user wrote something.

The messages are stored in memory.

Note

Note: If you don’t IRC very often and you’re on very active channels, this module might use up too much memory for you.

Chapter 9. socket module

Support for IPv4, IPv6 and pipes

Version: 0.1

Author: Jelmer Vernooij [mailto:]

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

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.

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

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.

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 10. noticelog module

Logging ctrlproxy messages via IRC

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://jelmer.vernstok.nl/ctrlproxy/

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 11. stats module

Stats generation

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://jelmer.vernstok.nl/ctrlproxy/
Requirements:	tdb, pcre(optional)

Description

This module keeps track of the number of times a certain expression is used and stores that data in a TDB-file. This TDB file can be read later with the **printstats** utility.

Expressions should be Perl-compatible regexes whenever pcre is found, normal regular expressions when only the POSIX regex functions are found. If no regex functionality was found, just 'plain' text should be specified.

Configuration

The following XML elements are supported:

tdbfile Location of TDB file to store statistics in. $HOME/.ctrlproxy-stats.tdb is used when none is specified.

pattern Pattern to search for and keep track of.

type 'Variable' to add up to whenever this pattern is used

Example configuration

```
<ctrlproxy>
<plugin autoload="1" file="stats">
<tdbfile>/home/jelmer/.ctrlproxy-stats.tdb</tdbfile>
<pattern type="happy">[:;]([-]*)[\)D]</pattern>
<pattern type="unhappy">[:;]([-]*)[\/\(]</pattern>
<pattern type="foul">(shit|damn|fuck)</pattern>
<pattern type="question">\?([^ ]*)$</pattern>
<pattern type="exclamation">!([^ ]*)$</pattern>
<pattern type="lines">(.*)</pattern>
<pattern type="word">([^ ]+)</pattern>
<pattern type="caps">^([^a-z]+)$</pattern>
<pattern type="action">.ACTION .</pattern>
<pattern type="violent">.ACTION .*(mept|slaat|kickt|kicks|duwt|slaps)</pattern>
</plugin>
<plugin autoload="1" file="socket"/>
</plugins>
<networks>
<network autoconnect="1">
<servers><ipv4 host="irc.freenode.net"/></servers>
<channel name="#flood" autojoin="1"/>
</network>
</networks>
</ctrlproxy>
```

Chapter 12. strip module

Strip query answers for other clients

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://jelmer.vernstok.nl/ctrlproxy/

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 13. Custom logging module

Logging in a predefined format

Version: 0.1

Author: Jelmer Vernooij [mailto:]

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

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.

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.

join

%c Name of the channel the user joins.

part

%c Name of the channel the user is leaving.

%m Comment

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.

quit

%m Comment.

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'.

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.

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.

nick

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

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.

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 14. nickserv module

NickServ

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://ctrlproxy.vernstok.nl/

Description

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

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 15. antiflood module

Flood protection module

Version:	0.1
Author:	Jelmer Vernooij [mailto:]
Homepage:	http://jelmer.vernstok.nl/ctrlproxy/

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.

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>
```

Part IV. Writing your own modules

Table of Contents

Chapter 16. 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 transport. 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).

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
```

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.

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.

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

Log functionality

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

Replication

The default replication function (default_replicate_function) of ctrlproxy is very basic. It only makes sure the client knows on which channels the user is, but does not do any replication of the messages that have been received.

To use your own replicate function instead, set the function pointer **replicate_function** to your replicate function.

The prototype for a replicate function is:

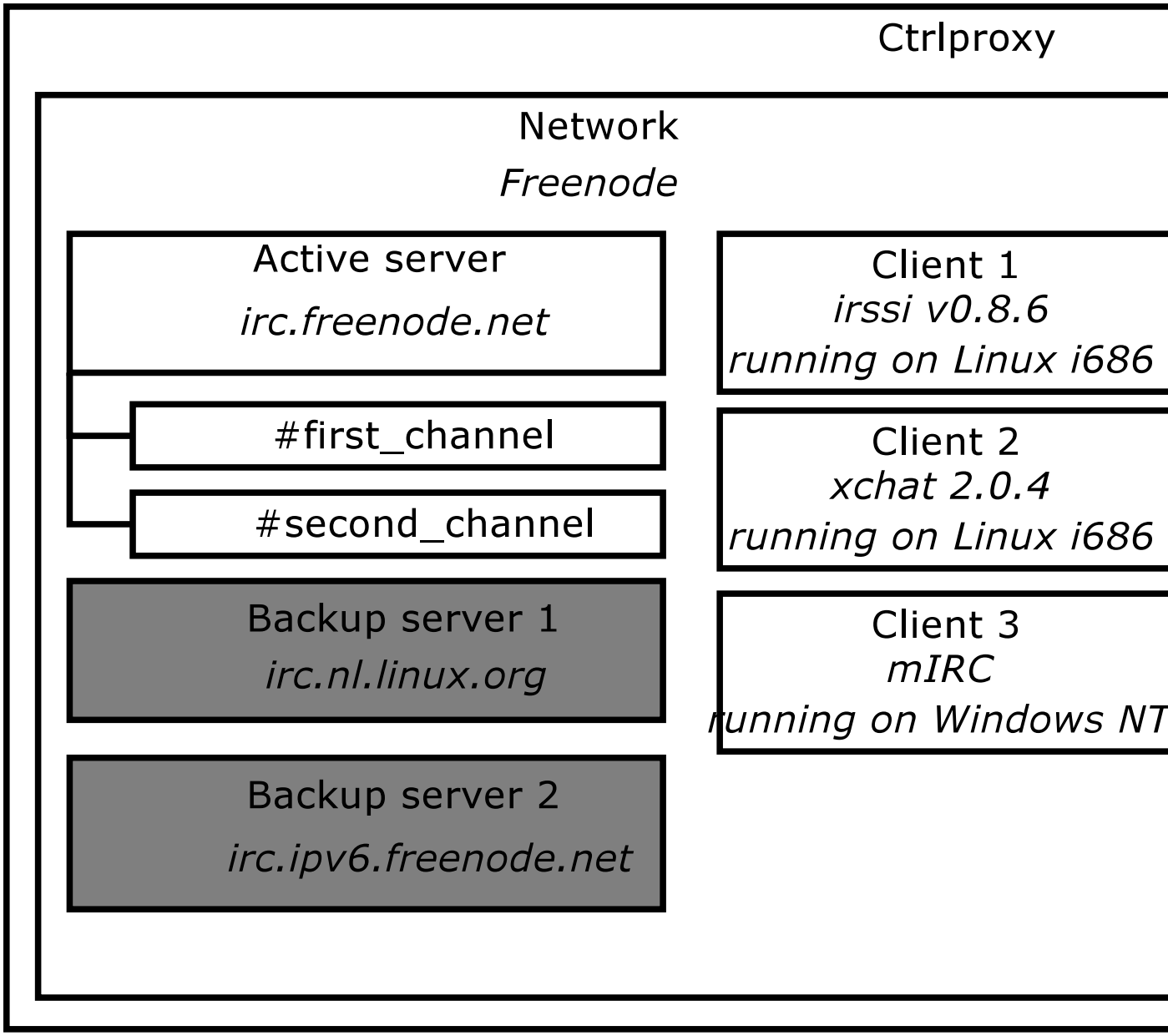
```
 void replicatef(struct network *s, struct transport_context *c);
```

The replication data should be sent to the specified transport_context.

Chapter 17. API

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

Main structs



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.

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.

State data

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

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.

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.

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.

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.

gen_replication()

```
GSList *gen_replication(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.

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.

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.

network_add_listen()

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

Add listener to specified network with configuration specified in xmlNodePtr.

save_configuration()

```
void save_configuration();
```

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

load_plugin()

```
void load_plugin(xmlNodePtr);
```

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

unload_plugin()

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

Try to unload the specified plugin. Not all plugins support this at the moment. If one attempts to unload a plugin that does not support unloading, ctrlproxy might crash.

Transports

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

register_transport()

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

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

transport_connect()

```
struct transport_context *transport_connect(const char *name, xmlNodePtr p,↩
receive_handler, disconnect_handler, void *data);
```

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.

transport_listen()

```
struct transport_context *transport_listen(const char *name, xmlNodePtr p,↩
newclient_handler, void *data);
```

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.

transport_free

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

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

transport_write()

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

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

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.

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.

transport_set_newclient_handler()

```
typedef void (*newclient_handler) (struct transport_context *, struct
transport_context *, void *data);
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.

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).

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*.

linedup()

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

Duplicate the given line struct.

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.

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.

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).

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.

free_line()

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

Free all data associated with *l*.

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.

irc_send_line()

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

Send the specified line to the specified transport_context.

clients_send()

```
void clients_send(struct network *, struct line *, struct transport_context↩
*exception);
```

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

General purpose functions

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.

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 18. 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*.

Transport contexts

The following struct is passed to all transport functions.

```
struct transport_context {
 struct transport *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.

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.

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.