

Methanol Web Crawling System

Manual for the Methanol Web Crawling System, version 1.7.0.

Emil Romanus

This manual is for Methanol Web Crawling System, version 1.7.0.

Copyright © 2008, 2009 Emil Romanus <sdac@bithack.se>

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Short Contents

1	Introduction	2
2	Methanol Programs	4
3	Installation	6
4	Post-Installation Setup and Testing	8
5	Configuration Files	11
6	Parsers	19
7	Scripting the Client	21
8	System Hook Scripts	26
9	Modules	28
10	Methanol Protocol	29
11	Copying	30
A	Robots Exclusion Standard	31
	Index	32

Table of Contents

1	Introduction	2
1.1	libmetha Feature Overview	2
1.2	Overview	2
2	Methanol Programs	4
2.1	mb – Methabot Command Line Tool	4
2.1.1	Invoking mb	4
2.1.2	mb command line options	4
2.1.2.1	Crawler Options	4
2.1.2.2	Filetype Options	5
2.1.2.3	General Options	5
2.2	mb-client – The Web Crawler Client	5
2.3	mn-masterd – The Master Server	5
2.4	mn-slaved – The Slave Server	5
2.5	madmind – Server Administration Tool	5
3	Installation	6
3.1	Building from source	6
3.1.1	Package dependencies	6
3.1.2	Running configure	6
3.1.3	Compiling and installing	7
3.2	Installing phpMadMind	7
4	Post-Installation Setup and Testing	8
4.1	Creating a MySQL Database and User	8
4.2	Example: Setup Master and Slave on a Single Server	9
4.2.1	Creating a System User	9
4.2.2	Directory Structure	9
4.2.3	Configuration	9
4.3	Starting and Troubleshooting	9
5	Configuration Files	11
5.1	Configuring mb and mb-client	11
5.1.1	Tutorial	11
5.1.1.1	Defining a filetype	11
5.1.1.2	Setting the parser	12
5.1.1.3	Creating your own crawler	13
5.1.2	Keywords	13
5.1.2.1	extend: Modify existing	13
5.1.2.2	override: Replace existing	14
5.1.2.3	copy: Copy existing to new	14
5.1.3	Advanced Topics	15

5.1.3.1	Crawler Switching	15
5.1.3.2	Handler Functions	15
5.2	<code>mn-masterd.conf</code> : Configuring the Master Server	16
5.2.1	<code>master</code> Option Reference	16
5.2.2	<code>user</code> Option Reference	17
5.2.3	<code>slave</code> Option Reference	17
5.3	<code>mn-slaved.conf</code> : Configuring the Slave Server	17
5.3.1	Option Reference	17
5.4	<code>mb-client.conf</code> : Configuring the Client Daemon	18
5.4.1	Option Reference	18
6	Parsers	19
6.1	Parser Chaining	19
6.2	List of built-in parsers	19
7	Scripting the Client	21
7.1	Global Functions	21
7.2	The <code>this</code> object	21
7.2.1	Member Functions	21
7.3	Tutorial: Writing a parser in Javascript with E4X	21
7.3.1	Getting Started	22
7.3.2	Extracting links using E4X	22
7.3.3	Extracting specific data	23
7.4	Init Functions	24
7.4.1	Writing an Init Function	24
7.4.2	Testing an Init Function	25
8	System Hook Scripts	26
8.1	Script Files	26
8.2	Preparing <code>mn-slaved</code>	26
8.3	Supported Hooks	27
8.3.1	<code>cleanup</code>	27
8.3.2	<code>session_complete</code>	27
9	Modules	28
9.1	Supported modules	28
9.1.1	<code>lmm_mysql</code> – Javascript-MySQL Bindings	28
9.1.2	<code>lmm_hash</code> – Functions for calculating checksums	28
9.1.3	<code>lmm_file</code> – C-like file and directory handling	28
9.2	Module C API	28
9.2.1	Example parser: Convert to lowercase	28
9.2.2	Creating a simple build system	28
9.2.3	Adding Javascript functions and classes	28
10	Methanol Protocol	29

11 Copying.....	30
Appendix A Robots Exclusion Standard.....	31
Index	32

This manual is for the Methanol Web Crawling System, version 1.7.0.

Understanding and running Methanol

Customization

Copying and license

Appendices

Indices

1 Introduction

Methanol is a complete web crawling system aiming for optimal customizability. Methanol tries to be more than just a web crawling system, and gives you the option to modify how and what data should be indexed, processed and ultimately displayed to the user.

Methanol's primary point is its web crawler Methabot. Methabot has been thoroughly tested on thousands of different kinds of websites and website layouts. Methabot is powered by its underlying web crawling library `libmetha`.

1.1 libmetha Feature Overview

- Speed-optimized architectural design
- Scriptable through Javascript with the E4X extension
- User-defined filetype filtering (according to MIME type, file extension or UMEX expression)
- Wide support for multi-threading (each thread is known as a worker)
- Extensible module system, supporting custom data parsers, filters and protocol handlers.
- Simple yet powerful filtering of URLs through UMEX.
- Support for automatic cookie handling when running over HTTP
- Robots Exclusion Standard
- Reliable, fault-tolerant networking, redirect-loop detection and some spider trap detection
- Parser chaining between different kinds of parsers, such as C and javascript parsers
- Simple and easy-to-use programming API
- HTML to XML/XHTML conversion
- Conversion between different character encodings, default encoding utf-8

1.2 Overview

There are a few concepts that you should be familiar with when working with Methanol systems. The probably most important thing to know is that a **crawler**, in terms of the Methanol system, is not a *function* or *process*. A crawler is merely a set of rules as to how web pages and files are crawled.

The actual process with threads crawling web pages or files should be referred to as a client with worker threads. Each worker may work independently on separate pages, but as long as they are running under the same process they share the same URL cache and set of rules.

A worker will always be directly connected to a single crawler at once. However, depending on how the configuration file looks, the worker may dynamically switch to another crawler.

The primary difference between any two crawlers is their lists of filetypes. One crawler might for example have filetype definitions for HTML files, while another crawler might

have filetype definitions for video files. The worker is restricted to its current crawler's list of filetypes.

When a worker is running with a specific crawler, it might find URLs or data matching one of the crawler's filetypes. The filetype in turn can have *parsers* and *attributes*. A parser is in short a script or callback function that sets attributes. As an example, you could define a filetype named "HTML" with attributes such as "title" and "description". The parsers job is to extract data and set these attributes.

When a client is connected to a Methanol system, data can be uploaded to the system if the attributes talked about above were set by a parser. Once the data has been uploaded, they will be available in the MySQL database.

2 Methanol Programs

This chapter will introduce you to the various programs included in the Methanol suite.

2.1 mb – Methabot Command Line Tool

mb is short for Methabot. Methabot is a handy command line tool for fetching and extracting data from the web or local files.

2.1.1 Invoking mb

Get a list of available runtime options by invoking:

```
$ mb --help
```

Another useful command to know is:

```
$ mb --info
```

The above example will output a list of runtime information about the installed version of Methabot. Most interesting is the list of default configuration files that are installed.

To load and use any default configuration file, a colon prefix is used. As an example, to load "archive", run:

```
$ mb :archive
```

Merely loading a configuration will not do anything useful, but a URL must be provided. A URL can be provided on command line, any argument not beginning with "-" or ":" will be assumed to be a URL.

```
$ mb :archive www.gnome.org
```

The above command will download and parse the front page of gnome.org, and print a list of all archive files found.

This behaviour is generally the default behaviour of methabot, provide it with one configuration and one URL, and it will return a list of target URLs.

2.1.2 mb command line options

The easiest way of changing Methabot's behaviour is through command line. This way of configuring methabot is not as flexible as by writing a configuration file, but it is still very powerful and most importantly easy.

The following three sections will describe the three different kinds of command line configuration options available in Methabot.

2.1.2.1 Crawler Options

A crawler option affects how the crawling is performed, think of it as behaviour options. Methabot (and libmetha in general) is capable of having multiple crawlers defined, and dynamically switching between crawlers at run time. The concept of crawler switching is however not covered in this chapter.

When configuring from command line, you will only be able to configure and change one crawler. By default, this will be the "default" crawler.

2.1.2.2 Filetype Options

Filetype options are used to define your own target filetype. Unlike crawler options, filetype options does not affect an already defined filetype, unless you use the `--filetype` option.

2.1.2.3 General Options

These options affect the runtime configuration in general, such as how many workers (see [Chapter 1 \[Introduction\], page 2](#)) to launch, proxy server settings or the user agent to use.

2.2 mb-client – The Web Crawler Client

The Methabot System Client (`mb-client`) is very similar to `mb`, but instead of communication with the user, `mb-client` communicates with a Methanol system.

`mb-client` can not be configured directly. It will always receive its configuration from a master server. `mb-client` only requires an IP-address to a master server when it is started.

In order to give `mb-client` the login credentials to the master server, it does however require a very minimal configuration file. This file will usually be locate at `/etc/mb-client.conf` depending on where you have installed Methanol. You can also use a custom configuration file by providing it from command line.

If a default configuration file does not exist, `mb-client` will attempt to connect to a master server running on `127.0.0.1:5505`, username *default* and password *default*.

2.3 mn-masterd – The Master Server

`mn-masterd`, also known as the master server, is the heart of a Methanol system. The purpose of the master is to distribute all connected clients to slave servers, and to keep statistics and configuration settings. The master is only required during system startup and when connecting new slaves or clients to the system. Once the system is up and running, no master is required.

When the master has been started it will listen for incoming connections, by default on port 5505, but that can be changed.

`mn-masterd` loads its configuration settings from `mn-masterd.conf`. Normally this file is located at `/etc/mn-masterd.conf`. It is also possible to give it another `mn-masterd.conf` using the `--config` command line option.

Start `mn-masterd` by running the following command:

```
$ mn-masterd
```

`mn-masterd` will fork and run in the background. You can stop it using `madmind` or by sending it the INT signal.

2.4 mn-slaved – The Slave Server

`mn-slaved` is the layer between the client and the database. `mn-slaved` manages clients, sessions, logging and runs system script hooks.

2.5 madmind – Server Administration Tool

`madmind` is a command line tool for communicating with the Master server. It is not available in this release, `phpMadMind` can be used instead.

3 Installation

Methanol is a part of the `methabot` package. The latest `methabot` packages includes all the source code required to build any specific part of the system you should want.

3.1 Building from source

This section will describe how to compile and install Methanol. During this section of the manual, you will get to choose parts and configure your own installation depending on what programs in the Methanol suite you need.

3.1.1 Package dependencies

To compile Methanol successfully, you will need the following package dependencies installed on your system, depending on which parts you plan to install:

Package	client-side ¹	server-side ²
MySQL (libmysqlclient) >= 5.0	no	yes
libcurl >= 7.16.0	yes	no
SpiderMonkey >= 1.7.0	yes	no
libev	<i>only mb-client</i>	yes
pthread	yes	yes

3.1.2 Running configure

Extract `methabot-1.7.0.tar.gz` and enter the created directory using the following commands:

```
$ tar xzf methabot-1.7.0.tar.gz
$ cd methabot-1.7.0
```

Now its time to configure what parts of the system you would like to install. For example, if you are installing a server you should most likely want to install `mn-slaved` and/or `mn-masterd` only, and `mb-client` on several other client computers. Or if you are only interested in the command line utility `mb`, you don't want to compile the server daemons.

By default, the configure script will configure the command line utility only, and the server daemons will not be compiled. The reason why they are not compiled by default is because they are still experimental and will be moved to a separate package in the next release.

```
$ ./configure
```

The above command will configure methabot for installing the command line tool `mb` only.

Configure for compiling and installing `mn-masterd` and `mn-slaved` only using the following command:

¹ Those programs counted as client-side are `mb` and `mb-client`

² Those programs counted as server-side are `mn-masterd` and `mn-slaved`

```
$ ./configure --enable-slave --enable-master --disable-cli
```

Notice the `--disable-cli` option, by disabling the compilation of the command line utility you don't have to have dependencies such as SpiderMonkey installed if you only want the server part installed.

Configure for compiling and installing the client daemon using the following command:

```
$ ./configure --enable-client
```

Finally, to get a list of all available options, run the following command:

```
$ ./configure --help
```

3.1.3 Compiling and installing

Once you have configured the package to fit your needs, its time to compile the code. The two following commands will compile and install the parts of the system that you configured it to install. Note that you might need root privileges to invoke the second command:

```
$ make
```

```
$ make install
```

3.2 Installing phpMadMind

phpMadMind is an official extension to the Methanol system. It is a set of PHP scripts for administrating a Methanol system. You can find phpMadMind in the latest `methabot` package, under the `src/` directory.

To install phpMadMind, you will first need a web server and PHP installed. You will also need the PHP extension `simplexml` installed.

Move the whole phpMadMind tree to any directory reachable from the web server, and open up `config.example.php`. This file is pretty straight-forward and you should be able to configure it on your own. Once you are done, save it as `config.php` and navigate to the directory using a web browser.

You will be presented with a login screen, if you just installed your system there will be a default user login with username "default" and password "default".

4 Post-Installation Setup and Testing

This chapter will help you set up and test an initial installation of the server programs included in Methanol.

Installed on your system you should have two configuration files, `mn-masterd.example.conf` and `mn-slaved.example.conf`. Usually, they are installed to the `/etc` directory, but it might depend on how you invoked `configure` during the installation part. These two files are skeleton configuration files for running the system for the first time. Rename them so `mn-masterd` and `mn-slaved` can find them:

```
$ cd /etc
$ mv mn-masterd.example.conf mn-masterd.conf
$ mv mn-slaved.example.conf mn-slaved.conf
```

These two files require information so that both server daemons will be able to connect to the MySQL server. If you have a MySQL database and user ready, then you can fill that information in right now and skip the next section.

Security is always an important factor. Node authentication is used to prevent others from connecting a client or a slave server to your master server. When you start `mn-masterd` using the skeleton configuration, it will accept any client or slave authenticating with the username *default* and password *default*.

As long as you are testing the system locally and trust your local users, you should however be safe.

Refer to [Chapter 5 \[Configuration Files\]](#), page 11 for information about how to set up node authentication.

4.1 Creating a MySQL Database and User

To set up a database and user login for Methanol to use, first connect to the MySQL server, either through a web interface such as phpMyAdmin or from command line as shown below:

```
$ mysql --user root --password
```

Now there are three MySQL statements that we need to execute. First, you must create the actual database that Methanol uses, and secondly a user which Methanol will log in to MySQL using.

```
mysql> CREATE DATABASE 'methanol';
mysql> GRANT ALL PRIVILEGES ON 'methanol'.* to 'username'@'localhost'
      IDENTIFIED BY 'password';
mysql> FLUSH PRIVILEGES;
```

Substitute *methanol* with a database name of choice, *username* with a username of choice and *password* with a password of choice. Also, you might want to modify the *localhost* to accept connections from other locations than the local host, depending on where you will be running the Methanol system and where the MySQL server is located.

Use the information you provided here to fill in the options in `mn-masterd.conf` and `mn-slaved.conf`.

4.2 Example: Setup Master and Slave on a Single Server

In many cases it might be feasible to run all system nodes on a single server. This section will give an example of how to set up such an environment.

Please walk through the chapter on installing Methanol, and configure Methanol with both the slave and the master daemon, and optionally the client daemon as well.

This example system environment will also prepare the slave for executing hook scripts.

4.2.1 Creating a System User

Both `mn-slaved` and `mn-masterd` will change their user id to the id of the user "nobody". This prevents them from touching irrelevant parts of the system, if someone would find and exploit a bug in the system.

A custom user still provides the same security as "nobody", but with support for executing hook scripts.

This example will create one user account that will be shared by both the master and the slave, we'll call it "mn-example".

```
$ useradd mn-example -m -s /sbin/nologin
```

The users home directory, `/home/mn-example`, will be used to structure the various parts of the system.

4.2.2 Directory Structure

Here is how we will layout this example system:

```
+ /home/mn-example/
+-- config/ : Configuration files should be put here
+-- run/    : Run-time files will be generated here
+-- hooks/  : All hook scripts should be put here
$ cd /home/mn-example
$ mkdir config run hooks
$ chown mn-example:mn-example *
$ chmod 700 *
```

4.2.3 Configuration

Set the `exec_dir` slave option to `/home/mn-example/run`. `user` and `group` should be set to `mn-example`, the user we created (and its corresponding group).

4.3 Starting and Troubleshooting

Start the master server by invoking:

```
$ mn-masterd
```

The process will fork and run in the background. If an error occurs it will notify you and exit with a status code of 1.

If an error occurs and you think the error message reported in the terminal doesn't help you find the cause of the error, then check your system messages. Both `mn-masterd` and `mn-slaved` uses `syslog` to log error messages and warnings. Where these messages are stored depends on your `syslog` installation, but most likely you get the latest messages by running:

```
$ tail /var/log/messages
```

Once `mn-masterd` is up and running, you can try connecting `mn-slaved` to it. In this case, it is important that the `master_host` option in `mn-slaved.conf` matches the listening address of the master server.

`mn-slaved` will log its error and warning messages using syslog as well.

5 Configuration Files

This chapter will help you understand how configuration files are structured, how to create your own configuration files and modify existing.

5.1 Configuring `mb` and `mb-client`

This section will help you understand how to configure Methabot and `mb-client`. Please note that if you are configuring `mb-client`, the configuration file should be put on the same server as the active `mn-masterd`, and not on the local host running the instance of `mb-client`.

5.1.1 Tutorial

Currently there are two kinds of classes you can create objects from in configuration files; crawlers and filetypes. Crawlers specify crawling behaviour, while each filetype specify properties for different filetypes such as audio files or HTML files.

A basic configuration file needs at least one crawler and one filetype. The crawler should be named `default`, but the filetype can be named anything. It is also possible to define more than one crawler, but that topic will not be covered in this tutorial.

Some modules (see [Chapter 9 \[Modules\], page 28](#)), such as `lmm_mysql` (see [Chapter 9 \[Modules\], page 28](#)), register extra functionality. They do this by registering a so-called scope. A scope is similiar to a filetype or crawler object, but does not require a name.

5.1.1.1 Defining a filetype

The most basic filetype requires only a name, and can be defined like below:

```
filetype["example"]
{
}
```

This will actually create an object of the class "filetype", with the name "example".

An empty filetype declaration isn't of much use. We must provide it with information about how to actually match URLs. This can be done in various ways, but `extensions` and `mimetypes` are the two primary options you should be playing with:

```
filetype["example"]
{
    extensions = {"png", "jpg", "jpeg"};
    mimetypes = {"image/jpeg", "image/png"};
}
```

As you can see, both `mimetypes` and `extensions` take arrays. The above code defines a filetype matching png and jpeg files.

To try the filetype out, you must first include a default configuration. This is required because your current configuration does not define a default crawler. If you include "default.conf", you don't have to configure a crawler. Below is an example of including another configuration file:

```
include "default.conf"
```

```

filetype["example"]
{
    extensions = {"png", "jpg", "jpeg"};
    mimetypes = {"image/jpeg", "image/png"};
}

```

The `include` directive (see [Chapter 5 \[Configuration Files\], page 11](#)) literally inserts the contents of another configuration file at the position of the `include` directive in this file. Think of it as `@import` in CSS or `#include` in C.

The `default.conf` configuration will define a default crawler, along with its own filetypes for crawling HTML and text files. Hence, if you include `default.conf` and run with your configuration, Methabot will be able to crawl websites and concentrate on finding the filetype you have declared.

Now to try your configuration out, put the above code in a file named `example.conf`. Move `example.conf` to `~/methabot/` and run the following command:

```
$ mb :example anyurl.com/path/
```

Methabot will in the above case look for `example.conf` first in `~/methabot/` and then in the default installation path. In short, your custom configurations will override the installed configurations.

Unless the above command failed, you should see a list of all the jpeg and png files found on that URL. Use the `-D` option to specify the crawling depth:

```
$ mb :example anyurl.com/path/ -D 2
```

5.1.1.2 Setting the parser

The `parser` option is used by filetypes to bind a parser. When a URL matches this filetype, the parser will be called to extract URLs or meta data. For example, you can use the built-in parser "css" to extract URLs such as images from CSS files:

```

include "default.conf"

filetype["example"]
{
    extensions = {"png", "jpg", "jpeg"};
    mimetypes = {"image/jpeg", "image/png"};
}

filetype["css"]
{
    extensions = {"css"};
    parser = "css";
}

```

Since `default.conf` defined filetypes for HTML crawling, you now extended its functionality by also adding support for crawling CSS files. Try running with the above configuration and you should see that Methabot will crawl HTML as expected, but also follow links to CSS files and try to find png and jpeg files there.

5.1.1.3 Creating your own crawler

Before we get started you must know that a crawler is merely a configuration, and not related to multi-threading in any way. A thread in libmetha is known as a worker, and workers can dynamically switch between different crawler configurations at any time. A crawler tells a worker exactly how to crawl a website, and what filetypes to look for.

Creating a custom crawler is more complex than creating a filetype. That is why the `extend` keyword is available. Using `extend`, you can modify the default crawler instead of creating your own from scratch:

```
include "default.conf"

extend: crawler["default"]
{
}
```

Inside the brackets you should set all variables you would like to modify from their default values. A tip is to have a look at the default configuration files and learn from them by example.

To modify the default depth limit of this crawler:

```
include "default.conf"

extend: crawler["default"]
{
    depth_limit = 2;
}
```

5.1.2 Keywords

5.1.2.1 `extend`: Modify existing

The `extend` keyword lets you modify an already defined object, and thus "extend" its attributes. This keyword is useful when you for example include a default configuration file just to modify a tiny setting. Here is an example:

```
include "default.conf"

extend: crawler["default"]
{
    dynamic_url = "discard";
}

extend: filetype["html"]
{
    parser = "blah";
}
```

The above will modify the crawler "default" defined in `default.conf`, and also change the parser of the filetype "html".

5.1.2.2 override: Replace existing

The `override` keyword works just like `extend`, except for the detail that it clears all settings in the target object before modifying it, and thus it overrides its definition completely. Example:

```
include "default.conf"

override: filetype["html"]
{
    extensions = {"html"};
    parser = "example";
}
```

5.1.2.3 copy: Copy existing to new

Use this keyword to copy the settings from another filetype/crawler and base your object on those settings. Example:

```
include "default.conf"

filetype["html_2" copy "html"]
{
    /* this defines the filetype "html_2" as a copy of "html" */
    extensions = {"example"};
    /* html_2 will now be identical to HTML, except for the extensions
    * array which was explicitly set */
}
```

You can also combine the `copy` keyword with `extend` or `override`. Though note that even if you combine `copy` with `extend`, the result will be as if you combined `copy` with `override`, since `copy` replaces all empty values as well. Here is an example combining `copy` and `override`:

```
include "image.conf"
include "audio.conf"

override: filetype["image" copy "audio"]
{
    /* 'image' is defined in image.conf, but since we copied 'audio'
    * to it, the image filetype will now be identical to the audio
    * filetype. Of course this does not make sense, it's just to
    * demonstrate. */

    mimetypes = {"image/jpeg"};
    /* the 'image' filetype now requires audio file extensions with
    * image file mimetypes! */
}
```

You can also simply copy filetypes to others, without modifying values, this can be pretty useful:

```
include "default.conf"
```


A handler can, just like a parser, be either a C or a Javascript function.

5.2 mn-masterd.conf: Configuring the Master Server

`mn-masterd` is configured through `mn-masterd.conf`, normally this file can be found at `/etc/mn-masterd.conf`. The layout of this file is simple, all options should be put inside a scope named `master`, like this:

```
master {
    listen = "127.0.0.1";
}
```

The above file will configure the listening address for the server.

Since `mn-masterd` will require both `mn-slaved` and `mb-client` to authenticate once they have connected, you will of course need to set authentication credentials.

To define a slave named "default", with password set to "pwd", allowed to log in from the local network:

```
slave["default"]
{
    password = "pwd";
    allow = {"192.168.1.0/24"};
}
```

The `allow` option defines where the slave is allowed to connect from. This option expects an array of subnets, that is network addresses in combination with subnet masks. Note that if you omit `allow` completely, all incoming connections will be accepted (assuming they login using the right password).

5.2.1 master Option Reference

<code>listen</code>	Listen address (and port). Set to "host:port" or "host".
<code>config_file</code>	The global system configuration file. This file will be sent to all connected clients, it should define crawlers and filetypes.
<code>session_complete_hook</code>	Script to run when a session is completed.
<code>cleanup_hook</code>	Hook for when the slave exits. See Chapter 8 [System Hook Scripts] , page 26.
<code>user</code>	Username of a local system user, the daemon will change its user id for security reasons. The value of this option should most likely be "nobody".
<code>group</code>	Username of a local system group. The value of this option should most likely be "nobody".
<code>mysql_host</code>	The host name or IP address of the MySQL server.
<code>mysql_user</code>	MySQL user name.

`mysql_pass`

MySQL user password.

`mysql_db` The database to select. Make sure the user has full privileges to the database.

5.2.2 user Option Reference

5.2.3 slave Option Reference

5.3 mn-slaved.conf: Configuring the Slave Server

`mn-slaved` loads its configuration file from `mn-slaved.conf`, which normally resides at `/etc/mn-slaved.conf`.

5.3.1 Option Reference

`listen` Listen address (and port). Set to "host:port" or "host".

`master_host`

Host IP address of the master server. If left unset, 127.0.0.1 is used.

`master_port`

Optional port number for the master server specified by `master_host`. If left unset, the default port number 5505 is used.

`master_user`

Username to use when login in to the master server. Corresponds to a `slave []` definition in `mn-masterd.conf`. If left unset, `default` is used.

`master_password`

Password to use when login in using the username specified with `master_user`. If left unset, `default` is used.

`user`

Username of a local system user, the daemon will change its user id for security reasons. The value of this option should most likely be "nobody", or a user you have specifically created for running `mn-slaved` as.

`group`

Username of a local system group. The value of this option should most likely be "nobody", or a group you have specifically created for running `mn-slaved` as.

`exec_dir`

If you have set up system hook scripts, the slave will download them to this directory before executing them. Make sure the user specified using `user` has permissions to execute and write to the directory.

`mysql_host`

The host name or IP address of the MySQL server.

`mysql_user`

MySQL user name.

`mysql_pass`

MySQL user password.

`mysql_db`

The database to select. Make sure the user has full privileges to the database.

5.4 mb-client.conf: Configuring the Client Daemon

`mb-client.conf` is used to set up login credentials for the client daemon to use when logging in to the master server. Configuration such as filetypes and crawlers can not be set in this file, but they will be received from the master server once connected.

If `mb-client.conf` can not be found by the daemon, it will assume the IP address of the master to be 127.0.0.1, the port 5505, username *default* and password *default*.

5.4.1 Option Reference

`master_host`

Host IP address of the master server. If left unset, 127.0.0.1 is used.

`master_port`

Optional port number for the master server specified by `master_host`. If left unset, the default port number 5505 is used.

`master_username`

Username to use when logging in to the master server. Corresponds to a `slave []` definition in `mn-masterd.conf`. If left unset, `default` is used.

`master_password`

Password to use when logging in using the username specified with `master_user`.

6 Parsers

The concept of *parsers* is one of the most important in Methanol. A parser is responsible for extracting URLs, meta-data and settings attributes for each retrieved URL.

Currently, a parser can be programmed in either C or Javascript. Multiple parsers can be chained and share the same data buffer.

6.1 Parser Chaining

Parser chaining is a concept introduced in libmetha-1.6.0. Parser chaining allows multiple parsers to work on the same data and share their modifications.

```

data -> [ parser_1 -> parser_2 -> parser_3 ] -> output_data
      |           |           |
      v           v           v
+-----+
|           List of found URLs           |
+-----+

```

If `parser_1` modified the data, `parser_2` will receive the modified version, and so on. Parser chaining works with any type of parser, thus you can freely use either javascript or C parsers anywhere in the chain.

To set a filetype to use a parser chain, you should provide it with a comma-separated list of parsers instead of just a single name of a parser. As an example, if you have created your own javascript parser that uses E4X to extract some metadata about a web page, you should most likely want to send the data (HTML) to the `xmlconv` builtin parser before your javascript parser receives it, to avoid XML errors caused by the HTML.

```

filetype["your_filetype"]
{
    parser = "xmlconv, yourfile.js/yourparser";
}

```

Furthermore, if your parser does not extract URLs but only extracts meta-information about the page, you can send it to the default HTML parser afterwards, which will extract all URLs for you:

```

filetype["your_filetype"]
{
    parser = "xmlconv, yourfile.js/yourparser, html";
}

```

6.2 List of built-in parsers

The following table lists the built-in parsers bundled with this version of `libmetha`.

<code>html</code>	The default HTML parser, designed for speed and fault-tolerance. Extracts URLs and adds them to the queue. If a <code><style></code> tag is encountered, the <code>css</code> parser is invoked on that piece. The same applies to plain text in the HTML source, which is forwarded to the <code>text</code> parser.
<code>text</code>	The <code>text</code> parser extracts URLs by searching for strings identifying the start of a URL, such as <code>http://</code> .

- css** The `css` parser is primarily intended for when image files are to be extracted or matched against a filetype. The `css` parser will parse CSS declarations, searching for anything that might be URL. `@import` is supported.
- xmlconv** Convert HTML/XHTML to valid XML. This parser is primarily used before data is sent to a Javascript parser that uses E4X to parse the data.
- utf8conv** Identifies the character encoding by either looking at the `Content-Type` HTTP header field, or a `<meta>` HTML element, and then converts the full text to UTF-8. Should be used when connected to a Methanol system, and also before parsing data with Javascript parsers. Note that running `utf8conv` will not slow down the processing if the data is already UTF-8, so it's safe to always call `utf8conv`.
- entityconv** Convert SGML entities to their UTF-8 equivalent. Note that this function will always write UTF-8 characters to the buffer. Be on the safe side by first invoking `utf8conv`, and then `entityconv`.
- ftp** Parses FTP directory listings. When libmetha crawls FTP-sites, the directory listings will be sent in plain text to the parser chain. Use this parser to safely handle lots of FTP server's differing directory listing outputs.

7 Scripting the Client

Methabot can be scripted in Javascript with E4X. Scripting is useful for creating custom parsers and extracting data from complex XML.

7.1 Global Functions

Additionally to the standard Javascript functions, `libmetha` provides the following functions that you can use:

- `print (str)`
Print a message to the user, without an ending new-line character.
- `println (str)`
Print a message to the user, with an ending new-line character.
- `getc ()` Get a character from *stdin*.
- `exec (command)`
Execute `command` in the default shell. Returns *false* on error. Returns the full output (i.e. the data written to *stdout* by the child process) as a *string*.

7.2 The `this` object

`this` is an object available in all javascript functions called by a worker in `libmetha`. `this` is almost like a direct connection to the actual worker. From `this`, you can get information such as HTTP status codes and the current URL. A complete table of what information you can get from it is displayed below.

- `status_code`
A number value depending on the used protocol and what value the target server returned. For HTTP, this value should be 200 on success.
- `content_type`
A string identifying the content type of the data in `this.data`.
- `url`
The URL of the current page and the contents of `this.data`.
- `data`
The raw data as returned from the server or a previous parser in the parser chain.

7.2.1 Member Functions

- `set_attribute (name, value)`
Set the `name` attribute to `value` for the current URL.

7.3 Tutorial: Writing a parser in Javascript with E4X

E4X is a very easy-to-use but powerful and flexible extension to Javascript. This tutorial will help you get started with writing parsers for `libmeth` clients using Javascript and E4X. You should have some previous experience with Javascript or another procedural programming/scripting language before attempting this tutorial.

You should not attempt to understand this tutorial unless you are already familiar with configuration files, as otherwise you may have a hard time getting your script loaded into the client.

This tutorial will use `mb` for testing, but the same code and configuration files can be used for `mb-client` or any client linked with `libmetha`.

7.3.1 Getting Started

A loadable parser should be a normal javascript function expecting zero arguments. All necessary information you will need will be available in an object called *this*. Your parser should return an array of all URLs it finds in the data, or null if no URL was found. The following snippet demonstrates a skeleton parser that always returns the same URL:

```
function test_parser()
{
    println("It works!");
    return Array("http://google.com/");
}
```

While it is possible to return a string instead of an Array, it is good practice to always return an Array, even in cases when only one URL is to be returned.

See [Chapter 7 \[Scripting the Client\], page 21](#) for more information about `println`.

To test the above parser, first save it to a file, let us say `example.js`. Save `example.js` to the default script path. You can get the default script path by invoking `mb --info`, by default `mb` has its default script path set to `/usr/share/metha/scripts`, and also a user-specific script path at `~/.methabot/scripts/`. For a normal installation, save `example.js`, to `~/.methabot/scripts/`. Create a simple configuration file that sets your parser as the default:

```
include "default.conf"

extend: filetype["html"]
{
    parser = "example.js/test_parser";
}
```

Notice that when you set the `parser` option of the `filetype`, you must include the name of the javascript file and the name of the function, separated using a slash.

Test your parser by running `mb` with your configuration file. If it "does not work", then try adding the `-e` flag to `mb` so Methabot won't discard `google.com` because it is an external URL.

Now let us get back to *this*. *this* is an object directly connected to a worker in `libmetha`. A worker is a thread, so each thread will have its own *this* object when running javascript code. The *this* object contains information such as the current URL and the current data downloaded from that URL.

For a full explanation of the *this* object, see [Chapter 7 \[Scripting the Client\], page 21](#).

7.3.2 Extracting links using E4X

Since most websites are coded in HTML, we will have to convert them to XML. Otherwise, we can't process them using E4X. `libmetha` comes with a default parser named `xmlconv`

for converting HTML/XHTML to XML. To convert the data to XML before it reaches your parser, you will have to set up a parser chain:

```
include "default.conf"

extend: filetype["html"]
{
    parser = "xmlconv, example.js/test_parser";
}
```

The order of the two parsers is important. Now that the data is converted to XML, we'll modify `test_parser` a little:

```
function test_parser()
{
    var xml = new XML(this.data);
    return xml..a.@href;
}
```

`this.data` is the data downloaded from the URL in question. `xmlconv` will modify `this.data` before it is sent to your `test_parser`.

An XML object is created from `this.data`. Elements in the XML can be accessed in a query-like manner – have a look at the return statement. After `xml` we have two dots, think of them as an expression for "any element", or more specifically, "any depth" in the XML tree. Next we have the `a`, and so far we thus have something like "look for an `<a>` element in any depth of the XML tree". Finally, `@href` returns the `href` attribute of the element. Since there can be multiple matches, i.e. many `<a>` elements, an `XMLList` is returned. An `XMLList` is almost like an array.

An important point to note once we've come this far is that you do not have to worry about the URL syntax of what you return. If you return "blah" and the current URL is `http://www.metha-sys.org/`, then `libmetha` will read your return value as `http://www.metha-sys.org/blah/`. It will also remove anchors (i.e. "...#blah") and normalize the URL in various ways. You don't have to do any sorting either, just return all the URLs you find and `libmetha` will sort them into filetypes.

Back to the E4X, you can concat multiple `XMLList`s with the '+' operator:

```
function test_parser()
{
    var x = new XML(this.data);
    return x..a.@href + x..img.@src;
}
```

7.3.3 Extracting specific data

Now that we have talked about how to locate elements and how to access their attributes in E4X, let's move on to the next step; finding specific data. Sometimes it is preferred to extract one or several specific URLs from a specific part of the document. Consider the following HTML code:

```
<html>
<head>
    <title>Example</title>
```

```

</head>
<body>
  <div id="header">
    <h1>Example Website</h1>
    <a href="uninteresting">Uninteresting</a>
  </div>
  <div id="content">
    <h2>Interesting URLs</h2>
    <a href="test1">test1</a>
    <a href="test2">test2</a>
  </div>
</body>
</html>

```

Say we want to extract the two URLs in the div with ID *content*, we can use an E4X expression like this one:

```

function test_parser()
{
  var x = new XML(this.data);
  println("Title of current URL: " + x..title); /* just for fun */
  return x..div.@id == "content"..a.@href;
}

```

Don't forget to add `xmlconv` to your parser chain any time you want to parse HTML as XML.

7.4 Init Functions

Each crawler configuration can have its own *init function*. The init function will run before any crawling is started. The purpose of the init function is to interpret initial input arguments and convert them to URLs. To visualize what's meant with input arguments, consider the following Methabot command:

```
$ mb -p 2 -d google.com test.com example.com
```

In this case, the input arguments are three; *google.com*, *test.com* and *example.com*. Normally, these arguments are URLs. However, an init function interpret the input arguments in any way and build its own URLs from them.

By default, a default init function will be used. This default init function will convert the arguments you give it to full URLs. In the case above it would have been a matter of adding *http://* in front of the all three only. This section will explain how to override the default init function and roll your own.

7.4.1 Writing an Init Function

To demonstrate the init function's usefulness, we will write a simple init function that uses the arguments sent to it to build google search string.

The first thing you should know is that unlike parser functions, init functions expect an argument. This argument will be the array of input data, one element per argument on command line. Thus, you will have to loop through the array to "build" your URLs for each argument. Here is a simple example:

```
function init_example(args)
{
    var ret = Array();

    for (var x=0; x<args.length; x++)
        ret.push("http://www.google.com/search?q="+args[x]);

    return ret;
}
```

7.4.2 Testing an Init Function

To test an init function, the `init` crawler option is used:

```
crawler["..."]
{
    ...
    init = "js_file.js/init_example";
    ...
}
```

8 System Hook Scripts

This chapter will talk about hook scripts. Hook scripts are used in a Methanol system for customization.

The scripts must be put on the master server. The master server will then distribute the scripts to the slave servers. Finally, the slave server(s) will execute the scripts.

Depending on what hook you are writing a script for, the script will be executed when different events occur in the system. For example, the `cleanup` hook is executed before a slave exits, and the `session_complete` hook is executed when a client has uploaded information about a web page.

A system hook can be scripted in *any* scripting language, as long as the language is supported by the slave server. This manual will however concentrate on hooks scripted in PHP.

8.1 Script Files

To set up a hook, set the hook's corresponding `mn-masterd` option to the full path of the script file. The file can be placed anywhere on the system. Example:

```
master {
    ...
    session_complete_hook = "/home/example/example.php";
    ...
}
```

The file itself, in this case `example.php`, must *always* begin with a line telling the operating system which interpreter to use. This is the line you see in any script file on your system. For example, a PHP file might begin with:

```
#!/usr/bin/php
... script ...
```

A bash file might begin with:

```
#!/bin/bash
... script ...
```

But the full path to the interpreter depends on where it is installed on the system.

Please note that the scripts will be run on slave servers, and not on the master server, so even though you put the script on the master, you must make sure the path to the interpreter applies on the slave server(s).

8.2 Preparing `mn-slaved`

`mn-slaved` will save the scripts received from the master to one file per script. To do this, the slave must have write permissions to a directory in the system.

First we create a directory where `mn-slaved` will be working. You can choose to create it anywhere in the system, as long as that partition is not mounted with `noexec`, but in this manual we will create it in `/var/mn-slaved`.

```
$ mkdir /var/mn-slaved
$ useradd mn-slaved -s /sbin/nologin -d /dev/null
```

```
$ chown mn-slaved:mn-slaved /var/mn-slaved
```

Now we can set up `mn-slaved` to use our new system user when forking. Change the `user` and `group` options of in `mn-slaved.conf` to "mn-slaved" both.

The `exec_dir` option must also be set, set it to the directory you created. To summarize:

```
slave {  
    ...  
    user = "mn-slaved";  
    group = "mn-slaved";  
    exec_dir = "/var/mn-slaved";  
}
```

8.3 Supported Hooks

This section will describe each hook, how it will be invoked and when it will be invoked.

8.3.1 cleanup

The `cleanup` hook will be invoked before the slave exits. This hook can be useful for cleaning up custom data added to the database, which is no longer needed after the slave has disconnected.

See the `cleanup_hook mn-masterd` option.

8.3.2 session_complete

The `session_complete` hook will be invoked once a session completes. When a slave sends a URL to the client, a session is started. This session will last until the client is out of URLs again. Note that the definition of a session is dependant on how the clients' crawlers are configured, such as depth limit.

See the `session_complete_hook mn-masterd` option.

9 Modules

9.1 Supported modules

9.1.1 `lmm_mysql` – Javascript-MySQL Bindings

9.1.2 `lmm_hash` – Functions for calculating checksums

9.1.3 `lmm_file` – C-like file and directory handling

9.2 Module C API

9.2.1 Example parser: Convert to lowercase

9.2.2 Creating a simple build system

9.2.3 Adding Javascript functions and classes

10 Methanol Protocol

11 Copying

This manual is for Methanol Web Crawling System, version 1.7.0.

Copyright © 2008, 2009 Emil Romanus <sdac@bithack.se>

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Appendix A Robots Exclusion Standard

Web crawlers such as `mb` and `mb-client` can download a file named `robots.txt` from the root directory of a web server. This file will contain rules as to which pages or parts of the website the bot may access and not. This system is known as the robots exclusion standard.

Enabling support for `robots.txt` is recommended when doing heavy crawling. You can enable `robots.txt` from command line using:

```
$ mb --robotstxt
```

You can also enable it per-crawler from a configuration file:

```
crawler["example"]
{
    robotstxt = true;
}
```

`libmetha` has wide support for the robots exclusion standard. The following table lists which directives that are supported.

User-agent

This directive is used to restrict the rules to a specific web crawler with a user agent matching the given value. Some websites might choose to block one bot from crawling a specific path, but allow another bot.

```
User-agent: Googlebot
Disallow: /
```

```
User-agent: Methabot
Allow: /
```

Disallow This directive is used to disallow the web crawling from accessing a part of the website matching the given pattern. Example:

```
Disallow: /private-directory/
```

Allow Used in combination with `disallow` to allow access to a document or path inside the `disallow-pattern`.

```
Disallow: /private/
Allow: /private/public.html
```

Note that this is a non-standard extensions, though it is supported by most web crawlers.

Index

C

configuration	11
Copying	30

H

hooks	26
-------------	----

I

Installation	10
introduction	3

M

methanol, programs	4
Modules	28

P

Parsers	20
protocol	29

S

scripting	25
setup, testing	8