The Polipo Manual

Juliusz Chroboczek

Polipo is a caching web proxy designed to be used as a personal cache or a cache shared among a few users.

Table of Contents

1 Background

1.1 The web and HTTP

The web is a wide-scale decentralised distributed hypertext system, something that's obviously impossible to achieve reliably.

The web is a collection of *resources* which are identified by *URLs*, strings starting with `http://`. At any point in time, a resource has a certain value, which is called an *instance* of the resource.

The fundamental protocol of the web is HTTP, a simple request/response protocol. With HTTP, a client can make a request for a resource to a server, and the server replies with an *entity*, which is an on-the-wire representation of an instance or of a fragment thereof.

1.2 Proxies and caches

A proxy is a program that acts as both a client and a server. It listens for client requests and forwards them to servers, and forwards the servers' replies to clients.

An HTTP proxy can optimise web traffic away by *caching* server replies, storing them in memory in case they are needed again. If a reply has been cached, a later client request may, under some conditions, be satisfied without going to the source again.

In addition to taking the shortcuts made possible by caching, proxies can improve performance by generating better network traffic than the client applications would do.

Proxies are also useful in ways unrelated to raw performance. A proxy can be used to contact a server that is not directly accessible to the client, for example because there is a firewall in the way (see Section 3.10 [Parent proxies], page 13), or because the client and the server use different lower layer protocols (for example IPv4 and IPv6). Another common application of proxies is to modify the data sent to servers and returned to clients, for example by censoring headers that expose too much about the client's identity (see Section 3.3.2 [Censoring headers], page 8) or removing advertisements from the data returned by the server (see Section 3.8 [Forbidden], page 11).

Polipo is a caching HTTP proxy that was originally designed as a *personal* proxy, i.e. a proxy that is used by a single user or a small group of users. However, it has successfully been used by larger groups.

1.3 Latency and throughput

Most network benchmarks consider *throughput*, or the average amount of data being pushed around per unit of time. While important for batch applications (for example benchmarks), average throughput is mostly irrelevant when it comes to interactive web usage. What is more important is a transaction's median *latency*, or whether the data starts to trickle down before the user gets annoyed.

Typical web caches optimise for throughput — for example, by consulting sibling caches before accessing a remote resource. By doing so, they significantly add to the median latency, and therefore to the average user frustration.

Polipo was designed to minimise latency.

1.4 Network traffic

The web was developed by people who were interested in text processing rather than in networking and, unsurprisingly enough, the first versions of the HTTP protocol did not make very good use of network resources. The main problem in HTTP/0.9 and early versions of HTTP/1.0

was that a separate TCP connection ("virtual circuit" for them telecom people) was created for every entity transferred.

Opening multiple TCP connections has significant performance implications. Obviously, connection setup and teardown require additional packet exchanges which increase network usage and, more importantly, latency.

Less obviously, TCP is not optimised for that sort of usage. TCP aims to avoid network *congestion*, a situation in which the network becomes unusable due to overly aggressive traffic patterns. A correct TCP implementation will very carefully probe the network at the beginning of every connection, which means that a TCP connection is very slow during the first couple of kilobytes transferred, and only gets up to speed later. Because most HTTP entities are small (in the 1 to 10 kilobytes range), HTTP/0.9 uses TCP where it is most inefficient.

1.4.1 Persistent connections

Later HTTP versions allow the transfer of multiple entities on a single connection. A connection that carries multiple entities is said to be *persistent* (or sometimes *keep-alive*). Unfortunately, persistent connections are an optional feature of HTTP, even in version 1.1.

Polipo will attempt to use persistent connections on the server side, and will honour persistent connection requests from clients.

1.4.2 Pipelining

With persistent connections it becomes possible to *pipeline* or *stream* requests, i.e. to send multiple requests on a single connection without waiting for the replies to come back. Because this technique gets the requests to the server faster, it reduces latency. Additionally, because multiple requests can often be sent in a single packet, pipelining reduces network traffic.

Pipelining is a fairly common technique[1], but it is not supported by HTTP/1.0. HTTP/1.1 makes pipelining support compulsory in every server implementation that can use persistent connections, but there are a number of buggy servers that claim to implement HTTP/1.1 but don't support pipelining.

Polipo carefully probes for pipelining support in a server and uses pipelining if it believes that it is reliable. Polipo also deeply enjoys being pipelined at by a client[2].

1.4.3 Poor Man's Multiplexing

A major weakness of the HTTP protocol is its inability to share a single connection between multiple simultaneous transactions — to *multiplex* a number of transactions over a single connection. In HTTP, a client can either request all instances sequentially, which significantly increases latency, or else open multiple concurrent connections, with all the problems that this implies (see Section 1.4.1 [Persistent connections], page 2).

Poor Man's Multiplexing (PMM) is a technique that simulates multiplexing by requesting an instance in multiple segments; because the segments are fetched in independent transactions, they can be interleaved with requests for other resources.

Obviously, PMM only makes sense in the presence of persistent connections; additionally, it is only effective in the presence of pipelining (see Section 1.4.2 [Pipelining], page 2).

PMM poses a number of reliability issues. If the resource being fetched is dynamic, it is quite possible that it will change between segments; thus, an implementation making use of PMM needs to be able to switch to full-resource retrieval when it detects a dynamic resource.

Polipo supports PMM, but it is disabled it by default (see Section 3.7 [PMM], page 10).

---

[1] The X11 protocol fundamentally relies on pipelining. NNTP does support pipelining. SMTP doesn't, while ESMTP makes it an option. FTP does support pipelining on the control connection.

[2] Other client-side implementations of HTTP that make use of pipelining include Opera, Mozilla, APT (the package downloader used by Debian GNU/Linux) and LFTP.

1.5 Caching partial instances

A partial instance is an instance that is being cached but only part of which is available in the local cache. There are three ways in which partial instances can arise: client applications requesting only part of an instance (Adobe's Acrobat Reader plugin is famous for that), a server dropping a connection mid-transfer (because it is short on resources, or, surprisingly often, because it is buggy), a client dropping a connection (usually because the user pressed *stop*).

When an instance is requested that is only partially cached, it is possible to request just the missing data by using a feature of HTTP known as a *range* request. While support for range requests is optional, most servers honour them in case of static data (data that are stored on disk, rather then being generated on the fly e.g. by a CGI script).

Caching partial instances has a number of positive effects. Obviously, it reduces the amount of data transmitted as the available data needn't be fetched again. Because it prevents partial data from being discarded, it makes it reasonable for a proxy to unconditionally abort a download when requested by the user, and therefore reduces network traffic.

Polipo caches arbitrary partial instances in its in-memory cache. It will only store the initial segment of a partial instance (from its beginning up to its first hole) in its on-disk cache, though. In either case, it will attempt to use range requests to fetch the missing data.

1.6 Other requests

The previous sections pretend that there is only one kind of request in HTTP — the '`GET`' request. In fact, there are some others.

The '`HEAD`' request method retrieves data about an resource. Polipo does not normally use '`HEAD`', but will fall back to using it for validation it if finds that a given server fails to cooperate with its standard validation methods (see Section 4.1 [Cache transparency], page 16). Polipo will correctly reply to a client's '`HEAD`' request.

The '`POST`' method is used to request that the server should do something rather than merely sending an entity; it is usually used with HTML forms that have an effect[3]. The '`PUT`' method is used to replace an resource with a different instance; it is typically used by web publishing applications.

'`POST`' and '`PUT`' requests are handled by Polipo pretty much like '`GET`' and '`HEAD`'; however, for various reasons, some precautions must be taken. In particular, any cached data for the resource they refer to must be discarded, and they can never be pipelined.

Finally, HTTP/1.1 includes a convenient backdoor with the '`CONNECT`' method. For more information, please see Section 3.12 [Tunnelling connections], page 14.

Polipo does not currently handle the more exotic methods such as '`OPTIONS`' and '`PROPFIND`'.

[3] HTML forms should use the '`GET`' method when the form has no side-effect as this makes the results cacheable.

2 Running Polipo

2.1 Starting Polipo

By default, Polipo runs as a normal foreground job in a terminal in which it can log random "How do you do?" messages. With the right configuration options, Polipo can run as a daemon.

Polipo is run with the following command line:

```
$ polipo [ -h ] [ -v ] [ -x ] [ -c config ] [ var=val... ]
```

All flags are optional. The flag `-h` causes Polipo to print a short help message and to quit. The flag `-v` causes Polipo to list all of its configuration variables and quit. The flag `-x` causes Polipo to purge its on-disk cache and then quit (see Section 4.3.2 [Purging], page 18). The flag `-c` specifies the configuration file to use (by default `~/.polipo` or `/etc/polipo/config`). Finally, Polipo's configuration can be changed on the command line by assigning values to given configuration variables.

2.1.1 Configuration

There is a number of variables that you can tweak in order to configure Polipo, and they should all be described in this manual (see [Variable index], page 23). You can display the complete, most up-to-date list of configuration variables by using the `-v` command line flag or by accessing the "current configuration" page of Polipo's web interface (see Section 2.4.1 [Web interface], page 6). Configuration variables can be set either on the command line or else in the configuration file given by the `-c` command-line flag.

Configuration variables are typed, and `-v` will display their types. The type can be of one of the following:

- '`integer`' or '`float`': a numeric value;
- '`boolean`': a truth value, one of '`true`' or '`false`';
- '`tristate`': one of '`false`', '`maybe`' or '`true`';
- '`4-state`', one of '`false`', '`reluctantly`', '`happily`' or '`true`';
- '`5-state`', one of '`false`', '`reluctantly`', '`maybe`', '`happily`' or '`true`';
- '`atom`', a string written within double quotes '`"`');
- '`list`', a comma-separated list of strings;
- '`intlist`', a comma-separated list of integers and ranges of integers (of the form '*n*–*m*').

The configuration file has a very simple syntax. All blank lines are ignored, as are lines starting with a hash sign '`#`'. Other lines must be of the form

```
var = val
```

where *var* is a variable to set and *val* is the value to set it to.

It is possible to change the configuration of a running polipo by using the local configuration interface (see Section 2.4.1 [Web interface], page 6).

2.1.2 Running as a daemon

If the configuration variable `daemonise` is set to true, Polipo will run as a daemon: it will fork and detach from its controlling terminal (if any). The variable `daemonise` defaults to false.

When Polipo is run as a daemon, it can be useful to get it to atomically write its *pid* to a file. If the variable `pidFile` is defined, it should be the name of a file where Polipo will write its *pid*. If the file already exists when it is started, Polipo will refuse to run.

2.1.3 Logging

When it encounters a difficulty, Polipo will print a friendly message. The location where these messages go is controlled by the configuration variables `logFile` and `logSyslog`. If `logSyslog` is `true`, error messages go to the system log facility given by `logFacility`. If `logFile` is set, it is the name of a file where all output will accumulate. If `logSyslog` is `false` and `logFile` is empty, messages go to the error output of the process (normally the terminal).

The variable `logFile` defaults to empty if `daemonise` is false, and to '`/var/log/polipo`' otherwise. The variable `logSyslog` defaults to `false`, and `logFacility` defaults to '`user`'.

If `logFile` is set, then the variable `logFilePermissions` controls the Unix permissions with which the log file will be created if it doesn't exist. It defaults to 0640.

The amount of logging is controlled by the variable `logLevel`. Please see the file '`log.h`' in the Polipo sources for the possible values of `logLevel`.

Keeping extensive logs on your users browsing habits is probably a serere violation of their privacy. If the variable `scrubLogs` is set, then Polipo will scrub most, if not all, private information from its logs.

2.2 Configuring your browser

Telling your user-agent (web browser) to use Polipo is an operation that depends on the browser. Many user-agents will transparently use Polipo if the environment variable '`http_proxy`' points at it; e.g.

```
$ export http_proxy=http://localhost:8123/
```

Netscape Navigator, Mozilla, Mozilla Firefox, KDE's Konqueror and probably other browsers require that you configure them manually through their *Preferences* or *Configure* menu.

If your user-agent sports such options, tell it to use persistent connections when speaking to proxies, to speak HTTP/1.1 and to use HTTP/1.1 pipelining.

2.3 Stopping Polipo and getting it to reload

Polipo will shut down cleanly if it receives `SIGHUP`, `SIGTERM` or `SIGINT` signals; this will normally happen when a Polipo in the foreground receives a `^C` key press, when your system shuts down, or when you use the `kill` command with no flags. Polipo will then write-out all its in-memory data to disk and quit.

If Polipo receives the `SIGUSR1` signal, it will write out all the in-memory data to disk (but won't discard them), reopen the log file, and then reload the forbidden URLs file (see Section 3.8 [Forbidden], page 11).

Finally, if Polipo receives the `SIGUSR2` signal, it will write out all the in-memory data to disk and discard as much of the memory cache as possible. It will then reopen the log file and reload the forbidden URLs file.

2.4 The local web server

Polipo includes a local web server, which is accessible on the same port as the one the proxy listens to. Therefore, by default you can access Polipo's local web server as '`http://localhost:8123/`'.

The data for the local web server can be configured by setting `localDocumentRoot`, which defaults to `/usr/share/polipo/www/`. Setting this variable to '`""`' will disable the local server.

Polipo assumes that the local web tree doesn't change behind its back. If you change any of the local files, you will need to notify Polipo by sending it a `SIGUSR2` signal (see Section 2.3 [Stopping], page 5).

If you use polipo as a publicly accessible web server, you might want to set the variable `disableProxy`, which will prevent it from acting as a web proxy. (You will also want to set `disableLocalInterface` (see Section 2.4.1 [Web interface], page 6), and perhaps run Polipo in a *chroot* jail.)

2.4.1 The web interface

The subtree of the local web space rooted at '`http://localhost:8123/polipo/`' is treated specially: URLs under this root do not correspond to on-disk files, but are generated by Polipo on-the-fly. We call this subtree Polipo's *local web interface*.

The page '`http://localhost:8123/polipo/config?`' contains the values of all configuration variables, and allows setting most of them.

The page '`http://localhost:8123/polipo/status?`' provides a summary status report about the running Polipo, and allows performing a number of actions on the proxy, notably flushing the in-memory cache.

The page '`http://localhost:8123/polipo/servers?`' contains the list of known servers, and the statistics maintained about them (see Section 3.5 [Server statistics], page 9).

The pages starting with '`http://localhost:8123/polipo/index?`' contain indices of the disk cache. For example, the following page contains the index of the cached pages from the server of some random company:

```
http://localhost:8123/polipo/index?http://www.microsoft.com/
```

The pages starting with '`http://localhost:8123/polipo/recursive-index?`' contain recursive indices of various servers. This functionality is disabled by default, and can be enabled by setting the variable `disableIndexing`.

If you have multiple users, you will probably want to disable the local interface by setting the variable `disableLocalInterface`. You may also selectively control setting of variables, indexing and listing known servers by setting the variables `disableConfiguration`, `disableIndexing` and `disableServersList`.

3 Polipo and the network

3.1 Client connections

There are three fundamental values that control how Polipo speaks to clients. The variable `proxyAddress`, defines the IP address on which Polipo will listen; by default, its value is the *loopback address* `"127.0.0.1"`, meaning that Polipo will listen on the IPv4 loopback interface (the local host) only. By setting this variable to a global IP address or to one of the special values `"::"` or `"0.0.0.0"`, it is possible to allow Polipo to serve remote clients. This is likely to be a security hole unless you set `allowedClients` to a reasonable value (see Section 3.1.1 [Access control], page 7).

Note that the type of address that you specify for `proxyAddress` will determine whether Polipo listens to IPv4 or IPv6. Currently, the only way to have Polipo listen to both protocols is to specify the IPv6 unspecified address (`"::"`) for `proxyAddress`.

The variable `proxyPort`, by default 8123, defines the TCP port on which Polipo will listen.

The variable `proxyName`, which defaults to the host name of the machine on which Polipo is running, defines the *name* of the proxy. This can be an arbitrary string that should be unique among all instances of Polipo that you are running. Polipo uses it in error messages and optionally for detecting proxy loops (by using the '`Via`' HTTP header, see Section 3.3.2 [Censoring headers], page 8). Finally, the `displayName` variable specifies the name used in user-visible error messages (default "Polipo").

3.1.1 Access control

By making it possible to have Polipo listen on a non-routable address (for example the loopback address '`127.0.0.1`'), the variable `proxyAddress` provides a very crude form of *access control*: the ability to decide which hosts are allowed to connect.

A finer form of access control can be implemented by specifying explicitly a number of client addresses or ranges of addresses (networks) that a client is allowed to connect from. This is done by setting the variable `allowedClients`.

Every entry in `allowedClients` can be an IP address, for example '`134.157.168.57`' or '`::1`'. It can also be a network address, i.e. an IP address and the number of bits in the network prefix, for example '`134.157.168.0/24`' or '`2001:660:116::/48`'. Typical uses of '`allowedClients`' variable include

```
allowedClients = 127.0.0.1, ::1, 134.157.168.0/24, 2001:660:116::/48
```

or, for an IPv4-only version of Polipo,

```
allowedClients = 127.0.0.1, 134.157.168.0/24
```

A different form of access control can be implemented by requiring each client to *authenticate*, i.e. to prove its identity before connecting. Polipo currently only implements the most insecure form of authentication, *HTTP basic authentication*, which sends usernames and passwords in clear over the network. HTTP basic authentication is required when the variable `authCredentials` is not null; its value should be of the form '`username:password`'.

Note that both IP-based authentication and HTTP basic authentication are insecure: the former is vulnerable to IP address spoofing, the latter to replay attacks. If you need to access Polipo over the public Internet, the only secure option is to have it listen over the loopback interface only and use an ssh tunnel (see Section 3.10 [Parent proxies], page 13)[1].

[1] It is not quite clear to me whether HTTP digest authentication is worth implementing. On the one hand, if implemented correctly, it appears to provide secure authentication; on the other hand, and unlike ssh or SSL, it doesn't make any attempt at ensuring privacy, and its optional integrity guarantees are impossible to implement without significantly impairing latency.

3.2 Contacting servers

A server can have multiple addresses, for example if it is *multihomed* (connected to multiple networks) or if it can speak both IPv4 and IPv6. Polipo will try all of a hosts addresses in turn; once it has found one that works, it will stick to that address until it fails again.

If connecting via IPv6 there is the possibility to use temporary source addresses to increase privacy (RFC 3041). The variable `useTemporarySourceAddress` controls the use of temporary addresses for outgoing connections; if set to `true` temporary addresses are preferred, if set to `false` static addresses are used and if set to `maybe` (the default) the operation system default is in effect. This setting is not available on all operation systems.

3.2.1 Allowed ports

A TCP service is identified not only by the IP address of the machine it is running on, but also by a small integer, the TCP *port* it is *listening* on. Normally, web servers listen on port 80, but it is not uncommon to have them listen on different ports; Polipo's internal web server, for example, listens on port 8123 by default.

The variable `allowedPorts` contains the list of ports that Polipo will accept to connect to on behalf of clients; it defaults to '`80-100, 1024-65535`'. Set this variable to '`1-65535`' if your clients (and the web pages they consult!) are fully trusted. (The variable `allowedPorts` is not considered for tunnelled connections; see Section 3.12 [Tunnelling connections], page 14).

3.3 Tuning at the HTTP level

3.3.1 Tuning the HTTP parser

As a number of HTTP servers and CGI scripts serve incorrect HTTP headers, Polipo uses a *lax* parser, meaning that incorrect HTTP headers will be ignored (a warning will be logged by default). If the variable `laxHttpParser` is not set (it is set by default), Polipo will use a *strict* parser, and refuse to serve an instance unless it could parse all the headers.

When the amount of headers exceeds one chunk's worth (see Section 5.1 [Chunk memory], page 20), Polipo will allocate a *big buffer* in order to store the headers. The size of big buffers, and therefore the maximum amount of headers Polipo can parse, is specified by the variable `bigBufferSize` (32 kB by default).

3.3.2 Censoring headers

Polipo offers the option to censor given HTTP headers in both client requests and server replies. The main application of this feature is to very slightly improve the user's privacy by eliminating cookies and some content-negotiation headers.

It is important to understand that these features merely make it slightly more difficult to gather statistics about the user's behaviour. While they do not actually prevent such statistics from being collected, they might make it less cost-effective to do so.

The general mechanism is controlled by the variable `censoredHeaders`, the value of which is a case-insensitive list of headers to unconditionally censor. By default, it is empty, but I recommend that you set it to '`From, Accept-Language`'. Adding headers such as '`Set-Cookie`', '`Set-Cookie2`', '`Cookie`', '`Cookie2`' or '`User-Agent`' to this list will probably break many web sites.

The case of the '`Referer`'[2] header is treated specially because many sites will refuse to serve pages when it is not provided. If `censorReferer` is `false` (the default), '`Referer`' headers are passed unchanged to the server. If `censorReferer` is `maybe`, '`Referer`' headers are passed to the

[2] HTTP contains many mistakes and even one spelling error.

server only when they refer to the same host as the resource being fetched. If `censorReferer` is `true`, all '`Referer`' headers are censored. I recommend setting `censorReferer` to `maybe`.

Another header that can have privacy implications is the '`Via`' header, which is used to specify the chain of proxies through which a given request has passed. Polipo will generate '`Via`' headers if the variable `disableVia` is `false` (it is true by default). If you choose to generate '`Via`' headers, you may want to set the `proxyName` variable to some innocuous string (see Section 3.1 [Client connections], page 7).

3.3.2.1 Why censor Accept-Language

Recent versions of HTTP include a mechanism known as *content negotiation* which allows a user-agent and a server to negotiate the best representation (instance) for a given resource. For example, a server that provides both PNG and GIF versions of an image will serve the PNG version to user-agents that support PNG, and the GIF version to Internet Explorer.

Content negotiation requires that a client should send with every single request a number of headers specifying the user's cultural and technical preferences. Most of these headers do not expose sensitive information (who cares whether your browser supports PNG?). The '`Accept-Language`' header, however, is meant to convey the user's linguistic preferences. In some cases, this information is sufficient to pinpoint with great precision the user's origins and even his political or religious opinions; think, for example, of the implications of sending '`Accept-Language: yi`' or '`ar_PS`'.

At any rate, '`Accept-Language`' is not useful. Its design is based on the assumption that language is merely another representation for the same information, and '`Accept-Language`' simply carries a prioritised list of languages, which is not enough to usefully describe a literate user's preferences. A typical French user, for example, will prefer an English-language original to a French (mis-)translation, while still wanting to see French language texts when they are original. Such a situation cannot be described by the simple-minded '`Accept-Language`' header.

3.3.3 Adjusting intermediate proxy behaviour

Implementors of intermediate caches (proxies) have found it useful to convert the media type of certain entity bodies. A non-transparent proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link.

If `alwaysAddNoTransform` is true (it is false by default), Polipo will add a 'no-transform' cache control directive to all outgoing requests. This directive forbids (compliant) intermediate caches from responding with an object that was compressed or transformed in any way.

3.4 Offline browsing

In an ideal world, all machines would have perfect connectivity to the network at all times and servers would never crash. In the real world, it may be necessary to avoid hitting the network and have Polipo serve stale objects from its cache.

Setting `proxyOffline` to `true` prevents Polipo from contacting remote servers, no matter what. This setting is suitable when you have no network connection whatsoever.

If `proxyOffline` is false, Polipo's caching behaviour is controlled by a number of variables documented in Section 4.1.2 [Tweaking validation], page 16.

3.5 Server statistics

In order to decide when to pipeline requests (see Section 1.4.2 [Pipelining], page 2) and whether to perform Poor Man's Multiplexing (see Section 1.4.3 [Poor Mans Multiplexing], page 2), Polipo needs to keep statistics about servers. These include the server's ability to handle persistent connections, the server's ability to handle pipelined requests, the round-trip time to the server,

and the server's transfer rate. The statistics are accessible from Polipo's web interface (see Section 2.4.1 [Web interface], page 6).

The variable '`serverExpireTime`' (default 1 day) specifies how long such information remains valid. If a server has not been accessed for a time interval of at least `serverExpireTime`, information about it will be discarded.

As Polipo will eventually recover from incorrect information about a server, this value can be made fairly large. The reason why it exists at all is to limit the amount of memory used up by information about servers.

3.6 Tweaking server-side behaviour

The most important piece of information about a server is whether it supports persistent connections. If this is the case, Polipo will open at most `serverSlots` connections to that server (`serverSlots1` if the server only implements HTTP/1.0), and attempt to pipeline; if not, Polipo will hit the server harder, opening up to `serverMaxSlots` connections.

Another use of server information is to decide whether to pipeline additional requests on a connection that already has in-flight requests. This is controlled by the variable `pipelineAdditionalRequests`; if it is `false`, no additional requests will be pipelined. If it is `true`, additional requests will be pipelined whenever possible. If it is `maybe` (the default), additional requests will only be pipelined following *small* requests, where a small request one whose download is estimated to take no more than `smallRequestTime` (default 5 s).

Sometimes, a request has been pipelined after a request that prompts a very large reply from the server; when that happens, the pipeline needs be broken in order to reduce latency. A reply is *large* and will cause a pipeline to be broken if either its size is at least `replyUnpipelineSize` (default one megabyte) or else the server's transfer rate is known and the body is expected to take at least `replyUnpipelineTime` to download (default 15 s).

The variable `maxPipelineTrain` defines the maximum number of requests that will be pipelined in a single write (default 10). Setting this variable to a very low value might (or might not) fix interaction with some unreliable servers that the normal heuristics are unable to detect.

The variable `maxSideBuffering` specifies how much data will be buffered in a PUT or POST request; it defaults to 1500 bytes. Setting this variable to 0 may cause some media players that abuse the HTTP protocol to work.

3.7 Poor Man's Multiplexing

By default, Polipo does not use Poor Man's Multiplexing (see Section 1.4.3 [Poor Mans Multiplexing], page 2). If the variable `pmmSize` is set to a positive value, Polipo will use PMM when speaking to servers that are known to support pipelining. It will request resources by segments of `pmmSize` bytes. The first segment requested has a size of `pmmFirstSize`, which defaults to twice `pmmSize`.

PMM is an intrinsically unreliable technique. Polipo makes heroic efforts to make it at least usable, requesting that the server disable PMM when not useful (by using the '`If-Range`' header) and disabling it on its own if a resource turns out to be dynamic. Notwithstanding these precautions, unless the server cooperates[3], you will see failures when using PMM, which will usually result in blank pages and broken image icons; hitting *Reload* on your browser will usually cause Polipo to notice that something went wrong and correct the problem.

[3] More precisely, unless CGI scripts cooperate.

3.8 Forbidden and redirected URLs

The web contains advertisements that a user-agent is supposed to download together with the requested pages. Not only do advertisements pollute the user's brain, pushing them around takes time and uses up network bandwidth.

Many so-called content providers also track user activities by using *web bugs*, tiny embedded images that cause a server to log where they are requested from. Such images can be detected because they are usually uncacheable (see Section 4.1 [Cache transparency], page 16) and therefore logged by Polipo by default.

Polipo can be configured to prevent certain URLs from reaching the browser, either by returning a *forbidden* error message to the user, or by *redirecting* such URLs to some other URL.

Some content providers attempt to subvert content filtering as well as malware scans by tunnelling their questionable content as https or other encrypted protocols. Other content providers are so clueless as to inject content from external providers into supposedly safe webpages. Polipo has therefore the ability to selectively block tunneled connections based on hostname and port information.

3.8.1 Internal forbidden list

The file pointed at by the variable `forbiddenFile` (defaults to `~/.polipo-forbidden` or `/etc/polipo/forbidden`, whichever exists) specifies the set of URLs that should never be fetched. If `forbiddenFile` is a directory, it will be recursively searched for files with forbidden URLs.

Every line in a file listing forbidden URLs can either be a domain name — a string that doesn't contain any of '`/`', '`*`' or '`\`' —, or a POSIX extended regular expression. Blank lines are ignored, as are those that start with a hash sign '`#`'.

By default, whenever it attempts to fetch a forbidden URL, the browser will receive a *403 forbidden* error from Polipo. Some users prefer to have the browser display a different page or an image.

If `forbiddenUrl` is not null, it should represent a URL to which all forbidden URLs will be redirected. The kind of redirection used is specified by `forbiddenRedirectCode`; if this is 302 (the default) the redirection will be marked as temporary, if 301 it will be a permanent one.

3.8.2 External redirectors

Polipo can also use an external process (a *Squid-style redirector*) to determine which URLs should be redirected. The name of the redirector binary is determined from the variable `redirector`, and the kind of redirection generated is specified by `redirectorRedirectCode`, which should be 302 (the default) or 301.

For example, to use Adzapper to redirect ads to an innocuous image, just set

```
redirector = /usr/bin/adzapper
```

3.8.3 Forbidden Tunnels

Polipo does by default allow tunnelled connections (see Section 3.12 [Tunnelling connections], page 14), however sometimes it is desirable to block connections selectively.

Because polipo does only pass through tunnelled connections filtering is possible based on hostname and port information only. Filtering based on protocol specific types of information like pathname is not possible.

Obviously the web browser (and other software) must be configured to use polipo as tunneling proxy for this to work. The tunnelled traffic is neither touched nor inspected in any way by

polipo, thus encryption, certification and all other security and integrity guarantees implemented in the browser are not in any way affected.

The file pointed at by the variable `forbiddenTunnelsFile` (defaults to `~/.polipo-forbiddenTunnels` or `/etc/polipo/forbiddenTunnels`, whichever exists) specifies the set of tunnel specifications that should be blocked.

Every line in a file listing forbidden Tunnels can either be a domain name — a string that doesn't contain any of '`/`', '`*`' or '`\`' —, or a POSIX extended regular expression. Blank lines are ignored, as are those that start with a hash sign '`#`'.

Entries in the form of regular expressions will be matched against tunnel reqeusts of the form `hostname:portnumber`.

Tunnelled and blocked connections will be logged if the configuration variable `logLevel` is set to a value such that `((logLevel & 0x80) !=0)`

Example `forbiddenTunnelsFile` :

```
# simple case, exact match of hostnames
www.massfuel.com

# match hostname against regexp
\.hitbox\.

# match hostname and port against regexp
# this will block tunnels to example.com but also  www.example.com
# for ports in the range 600-999
# Also watch for effects of 'tunnelAllowedPorts'
example.com\:[6-9][0-9][0-9]

# random examples
\.liveperson\.
\.atdmt\.com
.*doubleclick\.net
.*webtrekk\.de
^count\..*
.*\.offerstrategy\.com
.*\.ivwbox\.de
.*adwords.*
.*\.sitestat\.com
\.xiti\.com
webtrekk\..*
```

3.9 The domain name service

The low-level protocols beneath HTTP identify machines by IP addresses, sequences of four 8-bit integers such as '`199.232.41.10`'[4]. HTTP, on the other hand, and most application protocols, manipulate host names, strings such as '`www.polipo.org`'.

The *domain name service* (DNS) is a distributed database that maps host names to IP addresses. When an application wants to make use of the DNS, it invokes a *resolver*, a local library or process that contacts remote name servers.

Polipo usually tries to speak the DNS protocol itself rather than using the system resolver[5]. Its precise behaviour is controlled by the value of `dnsUseGethostbyname`. If

[4] Or sequences of eight 16-bit integers if you are running IPv6.

[5] The Unix interface to the resolver is provided by the `gethostbyname`(3) library call (`getaddrinfo`(3) on recent systems), which was designed at a time when a host lookup consisted in searching for one of five hosts in a

`dnsUseGethostbyname` is `false`, Polipo never uses the system resolver. If it is `reluctantly` (the default), Polipo tries to speak DNS and falls back to the system resolver if a name server could not be contacted. If it is `happily`, Polipo tries to speak DNS, and falls back to the system resolver if the host couldn't be found for any reason (this is not a good idea for shared proxies). Finally, if `dnsUseGethostbyname` is `true`, Polipo never tries to speak DNS itself and uses the system resolver straight away (this is not recommended).

If the internal DNS support is used, Polipo must be given a recursive name server to speak to. By default, this information is taken from the '`/etc/resolv.conf`' file; however, if you wish to use a different name server, you may set the variable `dnsNameServer` to an IP address[6].

When the reply to a DNS request is late to come, Polipo will retry multiple times using an exponentially increasing timeout. The maximum timeout used before Polipo gives up is defined by `dnsMaxTimeout` (default 60 s); the total time before Polipo gives up on a DNS query will be roughly twice `dnsMaxTimeout`.

The variable `dnsNegativeTtl` specifies the time during which negative DNS information (information that a host *doesn't* exist) will be cached; this defaults to 120 s. Increasing this value reduces both latency and network traffic but may cause a failed host not to be noticed when it comes back up.

The variable `dnsQueryIPv6` specifies whether to query for IPv4 or IPv6 addresses. If `dnsQueryIPv6` is `false`, only IPv4 addresses are queried. If `dnsQueryIPv6` is `reluctantly`, both types of addresses are queried, but IPv4 addresses are preferred. If `dnsQueryIPv6` is `happily` (the default), IPv6 addresses are preferred. Finally, if `dnsQueryIPv6` is `true`, only IPv6 addresses are queried.

If the system resolver is used, the value `dnsGethostbynameTtl` specifies the time during which a `gethostbyname` reply will be cached (default 5 minutes).

3.10 Parent proxies

Polipo will usually fetch instances directly from source servers as this configuration minimises latency. In some cases, however, it may be useful to have Polipo fetch instances from a *parent* proxy.

Polipo can use two protocols to speak to a parent proxy: HTTP and SOCKS. When configured to use both HTTP and SOCKS proxying, Polipo will contact an HTTP proxy over SOCKS — in other words, SOCKS is considered as being at a lower (sub)layer than HTTP.

3.10.1 HTTP parent proxies

The variable `parentProxy` specifies the hostname and port number of an HTTP parent proxy; it should have the form '`host:port`'.

If the parent proxy requires authorisation, the username and password should be specified in the variable `parentAuthCredentials` in the form '`username:password`'. Only *Basic* authentication is supported, which is vulnerable to replay attacks.

The main application of the parent proxy support is to cross firewalls. Given a machine, say `trurl`, with unrestricted access to the web, the following evades a firewall by using an encrypted compressed `ssh` link:

'`HOSTS.TXT`' file. The `gethostbyname` call is *blocking*, meaning that all activity must cease while a host lookup is in progress. When the call eventually returns, it doesn't provide a *time to live* (TTL) value to indicate how long the address may be cached. For these reasons, `gethostbyname` is hardly useful for programs that need to contact more than a few hosts. (Recent systems replace `gethostbyname`(3) by `getaddrinfo`(3), which is reentrant. While this removes one important problem that multi-threaded programs encounter, it doesn't solve any of the other issues with `gethostbyname`.)

[6] While Polipo does its own caching of DNS data, I recommend that you run a local caching name server. I am very happy with `pdnsd`, notwithstanding its somewhat bizarre handling of TCP connections.

```
$ ssh -f -C -L 8124:localhost:8123 trurl polipo
$ polipo parentProxy=localhost:8124
```

3.10.2 SOCKS parent proxies

The variable `socksParentProxy` specifies the hostname and port number of a SOCKS parent proxy; it should have the form '`host:port`'. The variant of the SOCKS protocol being used is defined by `socksProxyType`, which can be either '`socks4a`' or '`socks5`'; the latter value specifies "SOCKS5 with hostnames", and is the default.

The user name passed to the SOCKS4a proxy is defined by the variable `socksUserName`. This value is currently ignored with a SOCKS5 proxy.

The main application of the SOCKS support is to use Tor to evade overly restrictive or misconfigured firewalls. Assuming you have a Tor client running on the local host listening on the default port (9050), the following uses Tor for all outgoing HTTP traffic:

```
$ polipo socksParentProxy=localhost:9050
```

3.11 Tuning POST and PUT requests

The main assumption behind the design of the HTTP protocol is that requests are idempotent: since a request can be repeated by a client, a server is allowed to drop a connection at any time. This fact, more than anything else, explains the amazing scalability of the protocol.

This assumption breaks down in the case of POST requests. Indeed, a POST request usually causes some action to be performed (a page to be printed, a significant amount of money to be transferred from your bank account, or, in Florida, a vote to be registered), and such a request should not be repeated.

The only solution to this problem is to reserve HTTP to idempotent activities, and use reliable protocols for action-effecting ones. Notwithstanding that, HTTP/1.1 makes a weak attempt at making POST requests slightly more reliable and efficient than they are in HTTP/1.0.

When speaking to an HTTP/1.1 server, an HTTP client is allowed to request that the server check *a priori* whether it intends to honour a POST request. This is done by sending *an expectation*, a specific header with the request, '`Expect: 100-continue`', and waiting for either an error message or a '`100 Continue`' reply from the server. If the latter arrives, the client is welcome to send the rest of the POST request[7].

Polipo's behaviour w.r.t. client expectations is controlled by the variable `expectContinue`. If this variable is false, Polipo will never send an expectation to the server; if a client sends an expectation, Polipo will fail the expectation straight away, causing the client (if correctly implemented) to retry with no expectation. If `expectContinue` is `maybe` (the default), Polipo will behave in a standards-compliant manner: it will forward expectations to the server when allowed to do so, and fail client expectations otherwise. Finally, if `expectContinue` is `true`, Polipo will always send expectations when it is reasonable to do so; this violates the relevant standards and will break some websites, but might decrease network traffic under some circumstances.

3.12 Tunnelling connections

Polipo is an HTTP proxy; it proxies HTTP traffic, and clients using other protocols should either establish a direct connection to the server or use an *ad hoc* proxy.

In many circumstances, however, it is not possible to establish a direct connection to the server, for example due to mis-configured firewalls or when trying to access the IPv4 Internet

[7] This, of course, is only part of the story. Additionally, the server is not required to reply with '`100 Continue`', hence the client must implement a timeout. Furthermore, according to the obsolete RFC2068, the server is allowed to spontaneously send '`100 Continue`', so the client must be prepared to ignore such a reply at any time.

from an IPv6-only host. In such situations, it is possible to have Polipo behave as a *tunnelling* proxy — a proxy that merely forwards traffic between the client and the server without understanding it. Polipo enters tunnel mode when the client requests it by using the HTTP '`CONNECT`' method.

Most web browsers will use this technique for HTTP over SSL if configured to use Polipo as their 'https proxy'. More generally, the author has successfully used it to cross mis-configured firewalls using OpenSSH, rsync, Jabber, IRC, etc.

The variable `tunnelAllowedPorts` specifies the set of ports that Polipo will accept to tunnel traffic to. It defaults to allowing ssh, HTTP, https, rsync, IMAP, imaps, POP, pops, Jabber, CVS and Git traffic.

It is possible to selectively block tunneled connections, see Section 3.8.3 [Forbidden Tunnels], page 11

4 Caching

4.1 Cache transparency and validation

If resources on a server change, it is possible for a cached instance to become out-of date. Ideally, a cache would be perfectly *transparent*, meaning that it never serves an out-of-date instance; in a universe with a finite speed of signal propagation, however, this ideal is impossible to achieve.

If a caching proxy decides that a cached instance is new enough to likely still be valid, it will directly serve the instance to the client; we then say that the cache decided that the instance is *fresh*. When an instance is *stale* (not fresh), the cache will check with the upstream server whether the resource has changed; we say that the cached instance is being *revalidated*.

In HTTP/1.1, responsibility for revalidation is shared between the client, the server and the proxy itself. The client can override revalidation policy by using the '`Cache-Control`' header[1]; for example, some user-agents will request end-to-end revalidation in this way when the user shift-clicks on *reload*. The server may choose to specify revalidation policy by using the '`Expires`' and '`Cache-Control`' headers. As to the proxy, it needs to choose a revalidation policy for instances with neither server- nor client-side cache control information. Of course, nothing (except the HTTP/1.1 spec, but that is easily ignored) prevents a proxy from overriding the client's and server's cache control directives.

4.1.1 Tuning validation behaviour

Polipo's revalidation behaviour is controlled by a number of variables. In the following, an resource's *age* is the time since it was last validated, either because it was fetched from the server or because it was revalidated.

The policy defining when cached instances become stale in the absence of server-provided information is controlled by the variables `maxAge`, `maxAgeFraction`, `maxExpiresAge` and `maxNoModifiedAge`. If an instance has an '`Expires`' header, it becomes stale at the date given by that header, or when its age becomes larger than `maxExpiresAge`, whichever happens first. If an instance has no '`Expires`' header but has a '`LastModified`' header, it becomes stale when its age reaches either `maxAgeFraction` of the time since it was last modified or else the absolute value `maxAge`, whichever happens first. Finally, if an instance has neither '`Expires`' nor '`Last-Modified`', it will become stale when its age reaches `maxNoModifiedAge`.

4.1.2 Further tweaking of validation behaviour

If `cacheIsShared` is false (it is true by default), Polipo will ignore the server-side '`Cache-Control`' directives '`private`', '`s-maxage`' and '`proxy-must-revalidate`'. This is highly desirable behaviour when the proxy is used by just one user, but might break some sites if the proxy is shared.

When connectivity is very poor, the variable `relaxTransparency` can be used to cause Polipo to serve stale instances under some circumstances. If `relaxTransparency` is `false` (the default), all stale instances are validated (see Section 4.1 [Cache transparency], page 16), and failures to connect are reported to the client. This is the default mode of operation of most other proxies, and the least likely to surprise the user.

If `relaxTransparency` is `maybe`, all stale instances are still validated, but a failure to connect is only reported as an error if no data is available in the cache. If a connection fails and stale data is available, it is served to the client with a suitable HTTP/1.1 '`Warning`' header. Current user-agents do not provide visible indication of such warnings, however, and this setting will typically cause the browser to display stale data with no indication that anything went wrong.

[1] Or the obsolete '`Pragma`' header.

It is useful when you are consulting a live web site but don't want to be bothered with failed revalidations.

If `relaxTransparency` is `true`, missing data is fetched from remote servers, but stale data are unconditionally served with no validation. Client-side '`Cache-Control`' directives are still honoured, which means that you can force an end-to-end revalidation from the browser's interface (typically by shift-clicking on "reload"). This setting is only useful if you have very bad network connectivity or are consulting a very slow web site or one that provides incorrect cache control information[2] and are willing to manually revalidate pages that you suspect are stale.

If `mindlesslyCacheVary` is true, the presence of a '`Vary`' header (which indicates that content-negotiation occurred, see Section 3.3.2.1 [Censor Accept-Language], page 9) is ignored, and cached negotiated instances are mindlessly returned to the client. If it is false (the default), negotiated instances are revalidated on every client request.

Unfortunately, a number of servers (most notably some versions of Apache's `mod_deflate` module) send objects with a '`ETag`' header that will confuse Polipo in the presence of a '`Vary`' header. Polipo will make a reasonable check for consistency if '`dontTrustVaryETag`' is set to '`maybe`' (the default); it will systematically ignore '`ETag`' headers on objects with '`Vary`' headers if it is set to '`true`'.

A number of websites incorrectly mark variable resources as cachable; such issues can be worked around in polipo by manually marking given categories of objects as uncachable. If `dontCacheCookies` is true, all pages carrying HTTP cookies will be treated as uncachable. If `dontCacheRedirects` is true, all redirects (301 and 302) will be treated as uncachable. Finally, if everything else fails, a list of uncachable URLs can be given in the file specified by `uncachableFile`, which has the same format as the `forbiddenFile` (see Section 3.8.1 [Internal forbidden list], page 11). If not specified, its location defaults to '`~/.polipo-uncachable`' or '`/etc/polipo/uncachable`', whichever exists.

4.2 The in-memory cache

The in-memory cache consists of a list of HTTP and DNS objects maintained in least-recently used order. An index to the in-memory cache is maintained as a (closed) hash table.

When the in-memory cache grows beyond a certain size (controlled by a number of variables, see Chapter 5 [Memory usage], page 20), or when a hash table collision occurs, resources are written out to disk.

4.3 The on-disk cache

The on-disk cache consists in a filesystem subtree rooted at a location defined by the variable `diskCacheRoot`, by default `"/var/cache/polipo/"`. This directory should normally be writeable, readable and seekable by the user running Polipo. While it is best to use a local filesystem for the on-disk cache, a NFSv3- or AFS-mounted filesystem should be safe in most implementations. Do not use NFSv2, as it will cause cache corruption[3].

If `diskCacheRoot` is an empty string, no disk cache is used.

The value `maxDiskEntries` (32 by default) is the absolute maximum of file descriptors held open for on-disk objects. When this limit is reached, Polipo will close descriptors on a least-recently-used basis. This value should be set to be slightly larger than the number of resources that you expect to be live at a single time; defining the right notion of liveness is left as an exercise for the interested reader.

---

[2] This is for example the case of `www.microsoft.com`, and also of websites generated by a popular Free content management system written in Python.

[3] Polipo assumes that '`open(O_CREAT | O_EXCL)`' works reliably.

The value `diskCacheWriteoutOnClose` (64 kB by default) is the amount of data that Polipo will write out when closing a disk file. Writing out data when closing a file can avoid subsequently reopening it, but causes unnecessary work if the instance is later superseded.

The integers `diskCacheDirectoryPermissions` and `diskCacheFilePermissions` are the Unix filesystem permissions with which files and directories are created in the on-disk cache; they default to '`0700`' and '`0600`' respectively.

The variable `maxDiskCacheEntrySize` specifies the maximum size, in bytes, of an instance that is stored in the on-disk cache. If set to -1 (the default), all objects are stored in the on-disk cache,

4.3.1 Asynchronous writing

When Polipo runs out of memory (see Section 5.3 [Limiting memory usage], page 20), it will start discarding instances from its memory cache. If a disk cache has been configured, it will write out any instance that it discards. Any memory allocation that prompted the purge must then wait for the write to complete.

In order to avoid the latency hit that this causes, Polipo will preemptively write out instances to the disk cache whenever it is idle. The integer `idleTime` specifies the time during which Polipo will remain idle before it starts writing out random objects to the on-disk cache; this value defaults to 20 s. You may want to decrease this value for a busy cache with little memory, or increase it if your cache is often idle and has a lot of memory.

The value `maxObjectsWhenIdle` (default 32) specifies the maximum number of instances that an idle Polipo will write out without checking whether there's any new work to do. The value `maxWriteoutWhenIdle` specifies the maximum amount of data (default 64 kB) that Polipo will write out without checking for new activity. Increasing these values will make asynchronous write-out slightly faster, at the cost of possibly increasing Polipo's latency in some rare circumstances.

4.3.2 Purging the on-disk cache

Polipo never removes a file in its on-disk cache, except when it finds that the instance that it represents has been superseded by a newer version. In order to keep the on-disk cache from growing without bound, it is necessary to *purge* it once in a while. Purging the cache typically consists in removing some files, truncating large files (see Section 1.5 [Partial instances], page 3) or moving them to off-line storage.

Polipo itself can be used to purge its on-disk cache; this is done by invoking Polipo with the `-x` flag. This can safely be done when Polipo is running (see Section 4.3.4 [Modifying the on-disk cache], page 19).

For a purge to be effective, it is necessary to cause Polipo to write-out its in-memory cache to disk (see Section 2.3 [Stopping], page 5). Additionally, Polipo will not necessarily notice the changed files until it attempts to access them; thus, you will want it to discard its in-memory cache after performing the purge. The safe way to perform a purge is therefore:

```
$ kill -USR1 polipo-pid
$ sleep 1
$ polipo -x
$ kill -USR2 polipo-pid
```

The behaviour of the `-x` flag is controlled by three configuration variables. The variable `diskCacheUnlinkTime` specifies the time during which an on-disk entry should remain unused before it is eligible for removal; it defaults to 32 days.

The variable `diskCacheTruncateTime` specifies the time for which an on-disk entry should remain unused before it is eligible for truncation; it defaults to 4 days and a half. The variable

`diskCacheTruncateSize` specifies the size at which files are truncated after they have not been accessed for `diskCacheTruncateTime`; it defaults to 1 MB.

Usually, Polipo uses a file's modification time in order to determine whether it is old enough to be expirable. This heuristic can be disabled by setting the variable `preciseExpiry` to true.

4.3.3 Format of the on-disk cache

The on-disk cache consists of a collection of files, one per instance. The format of an on-disk resource is similar to that of an HTTP message: it starts with an HTTP status line, followed by HTTP headers, followed by a blank line ('`\r\n\r\n`'). The blank line is optionally followed by a number of binary zeroes. The body of the instance follows.

The headers of an on-disk file have a few minor differences with HTTP messages. Obviously, there is never a '`Transfer-Encoding`' line. A few additional headers are used by Polipo for its internal bookkeeping:

- '`X-Polipo-Location`': this is the URL of the resource stored in this file. This is always present.
- '`X-Polipo-Date`': this is Polipo's estimation of the date at which this instance was last validated, and is used for generating the '`Age`' header of HTTP messages. This is optional, and only stored if different from the instance's date.
- '`X-Polipo-Access`': this is the date when the instance was last accessed by Polipo, and is used for cache purging (see Section 4.3.2 [Purging], page 18). This is optional, and is absent if the instance was never accessed.
- '`X-Polipo-Body-Offset`': the presence of this line indicates that the blank line following the headers is followed by a number of zero bytes. Its value is an integer, which indicates the offset since the beginning of the file at which the instance body actually starts. This line is optional, and if absent the body starts immediately after the blank line.

4.3.4 Modifying the on-disk cache

It is safe to modify the on-disk cache while Polipo is running as long as no file is ever modified in place. More precisely, the only safe operations are to unlink (remove, delete) files in the disk cache, or to atomically add new files to the cache (by performing an exclusive open, or by using one of the '`link`' or '`rename`' system calls). It is *not* safe to truncate a file in place.

5 Memory usage

Polipo uses two distinct pools of memory, the *chunk pool* and the *malloc pool*.

5.1 Chunk memory

Most of the memory used by Polipo is stored in chunks, fixed-size blocks of memory; the size of a chunk is defined by the compile-time constant `CHUNK_SIZE`, and defaults to 4096 bytes on 32-bit platforms, 8192 on 64-bit ones. Chunks are used for storing object data (bodies of instances) and for temporary I/O buffers. Increasing the chunk size increases performance somewhat, but at the cost of larger granularity of allocation and hence larger memory usage.

By default, Polipo uses a hand-crafted memory allocator based on `mmap`(2) (`VirtualAlloc` under Windows) for allocating chunks; while this is very slightly faster than the stock memory allocator, its main benefit is that it limits memory fragmentation. It is possible to disable the chunk allocator, and use `malloc`(3) for all memory allocation, by defining `MALLOC_CHUNKS` at compile time; this is probably only useful for debugging.

There is one assumption made about `CHUNK_SIZE`: `CHUNK_SIZE` multiplied by the number of bits in an `unsigned long` (actually in a `ChunkBitmap` — see `chunk.c`) must be a multiple of the page size, which is 4096 on most systems (8192 on Alpha, and apparently 65536 on Windows).

As all network I/O will be performed in units of one to two chunks, `CHUNK_SIZE` should be at least equal to your network interface's MTU (typically 1500 bytes). Additionally, as much I/O will be done at `CHUNK_SIZE`-aligned addresses, `CHUNK_SIZE` should ideally be a multiple of the page size.

In summary, 2048, 4096, 8192 and 16384 are good choices for `CHUNK_SIZE`.

5.2 Malloc allocation

Polipo uses the standard `malloc`(3) memory allocator for allocating small data structures (up to 100 bytes), small strings and atoms (unique strings).

5.3 Limiting Polipo's memory usage

Polipo is designed to work well when given little memory, but will happily scale to larger configurations. For that reason, you need to inform it of the amount of memory it can use.

5.3.1 Limiting chunk usage

You can limit Polipo's usage of chunk memory by setting `chunkHighMark` and `chunkLowMark`.

The value `chunkHighMark` is the absolute maximum number of bytes of allocated chunk memory. When this value is reached, Polipo will try to purge objects from its in-memory cache; if that fails to free memory, Polipo will start dropping connections. This value defaults to 24 MB or one quarter of the machine's physical memory, whichever is less.

When chunk usage falls back below `chunkLowMark`, Polipo will stop discarding in-memory objects. The value `chunkCriticalMark`, which should be somewhere between `chunkLowMark` and `chunkHighMark`, specifies the value above which Polipo will make heroic efforts to free memory, including punching holes in the middle of instances, but without dropping connections.

Unless set explicitly, both `chunkLowMark` and `chunkCriticalMark` are computed automatically from `chunkHighMark`.

5.3.2 Limiting object usage

Besides limiting chunk usage, it is possible to limit Polipo's memory usage by bounding the number of objects it keeps in memory at any given time. This is done with `objectHighMark` and `publicObjectLowMark`.

The value `objectHighMark` is the absolute maximum of objects held in memory (including resources and server addresses). When the number of in-memory objects that haven't been superseded yet falls below `publicObjectLowMark`, Polipo will stop writing out objects to disk (superseded objects are discarded as soon as possible).

On 32-bit architectures, every object costs 108 bytes of memory, plus storage for every globally unique header that is not handled specially (hopefully negligible), plus an overhead of one word (4 bytes) for every chunk of data in the object.

You may also want to change `objectHashTableSize`. This is the size of the hash table used for holding objects; it should be a power of two and defaults to eight times `objectHighMark`. Increasing this value will reduce the number of objects being written out to disk due to hash table collisions. Every hash table entry costs one word.

5.3.3 OS usage limits

Many operating systems permit limiting a process' memory usage by setting a *usage limit*; on most Unix-like systems, this is done with the `-v` option to the `ulimit` command. Typically, the effect is to cause calls to the `malloc` and `mmap` library functions to fail.

Polipo will usually react gracefully to failures to allocate memory[1]. Nonetheless, you should avoid using OS limits to limit Polipo's memory usage: when it hits an OS limit, Polipo cannot allocate the memory needed to schedule recovery from the out-of-memory condition, and has no choice other than to drop a connection.

Unfortunately, some operating system kernels (notably certain Linux releases) fail to fail an allocation if no usage limit is given; instead, they either crash when memory is exhausted, or else start killing random processes with no advance warning[2]. On such systems, imposing an (unrealistically large) usage limit on Polipo is the safe thing to do.

[1] There are exactly three places in the code where Polipo will give up and exit if out of memory; all three are extremely unlikely to happen in practice.

[2] How I wish for a '`SIGXMEM`' signal.

Copying

You are allowed to do anything you wish with Polipo as long as you don't deny my right to be recognised as its author and you don't blame me if anything goes wrong.

More formally, Polipo is distributed under the following terms:

The last sentence is what happens when you allow lawyers to have it their way with a language.

Variable index

T

U

Concept index

R

S

T

U

V

W