The *POPular* Manual

Jochen Topf

The *POPular* Manual
by Jochen Topf

POPular is a suite of programs for running large mail storage servers with POP3 access.

Table of Contents

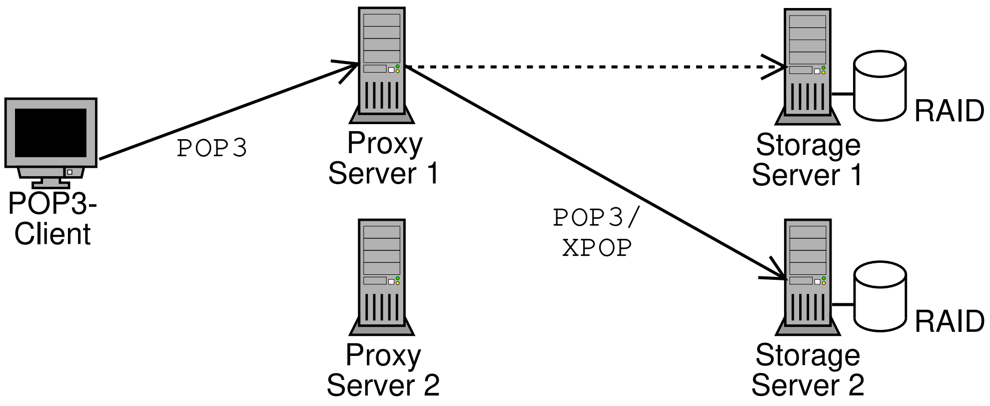
List of Figures

Introduction

A single server for mailbox storage, even if it is very powerful, can never scale to the number of mailboxes used in todays large mail systems with millions of mailboxes. So you need some way of distribution the mailboxes over several servers, while access for the client must be transparent.

The *POPular* POP server suite is one way of solving this problem. It implements a proxying architecture, where all POP access from the client is funneled through a group of proxies which authenticate the user, decide which storage server the mailbox is on and then connect the client with the proper backend server.

Figure 1. The basic *POPular* architecture.



1. Features

Scales to very large systems

The architecture of the *POPular* server system allows a mail system to scale to practically any size. There is no single point that every mail has to pass or any other bottlenecks. If the number of mailboxes grows beyond what your servers can handle, you can just add more servers and change the configuration a little bit.

Virtual Servers

The *POPular* proxy can be configured to accept connections to several IP numbers and/or TCP ports. The authentification function can use the information about the accessed IP number and port to decide which mailbox to access. An example: You can have two IP numbers for one proxy server, one with the name pop.server1.com, the other with the name pop.server2.com. A user with the username 'joe' on one server will be distinguishable from a user with the same name on the other server.

Optimization for empty mailboxes

On a typical POP server, about 80% of the accesses are to an empty mailbox. *POPular* is specially optimized for this case, reducing the load on the storage servers considerably.

Soft failure on many errors

On many errors, like failure of a backend server, the proxy will display an empty mailbox to the user, instead of an error message. When doing system maintainance the admin can configure all or a part of the system as 'disabled' and the user, again, sees an empty mailbox. This saves a lot of hassle for the user, the admin and the support staff.

Get information from running server

The admin can get detailed information about current connections, current configuration and some counters from the running proxy and server.

Configuration at runtime

The proxy and storage servers are fully configurable at runtime, there is no need to take the proxy down for anything other than software upgrades.

Verbose logging

The programs in the *POPular* server suite will log everything that happens, allowing you to diagnose problems rapidly. The log format is easily parsable to enable you to quickly write scripts for generating statistics or reacting to problems. Several types of debug logging can be enabled and disabled on the fly.

Proxying to POP3 servers or special XPOP server

The *POPular* proxy communicates on the one side with any standard complient POP3 client and on the other side with any standard POP3 server or with a special server included in the *POPular* suite talking the special XPOP protocol, which is basically a POP3 procotol without the authentication phase.

Fallback server

If the user can't be authenticated the proxy can fall back to any POP3 server. This is a nice feature for migrating to a new system.

Uses MailDir format

The *POPular* server uses Maildir format only, so there are no locking issues.

Flexible database access and authentication

Database access and authentication in the proxy is delegated to shared libraries loaded and configured at runtime. This allowes for a very flexible authentication mechanism.

Configurable maximum number of connections

The allowed number of connections can be configured to never overload the server.

SSL/TLS support

Starting in version 1.5.0 POPular supports SSL and TLS. Each virtual server can be configured individually to use SSL/TLS (with or without STARTTLS) and can have it's own key.

2. Definitions

Here are the definitions of some words used in this manual:

proxy (server)

This is the host that the mail client connects to. This server is running the **pproxy** program. It is responsible for authenticating the user, deciding on which backend server the mailbox of the user is located and what its name is. The proxy will then connect to the backend and forward data between client and backend server.

backend (server)

This is any kind of POP3 or storage server that will be accessed by the proxy server.

storage server

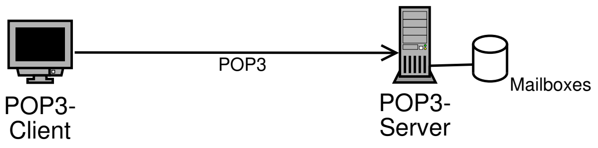
This is a backend server that has the *POPular* **pserv** running. It is only a place to store the messages.

Chapter 1. The *POPular* system architecture

1.1. Overview

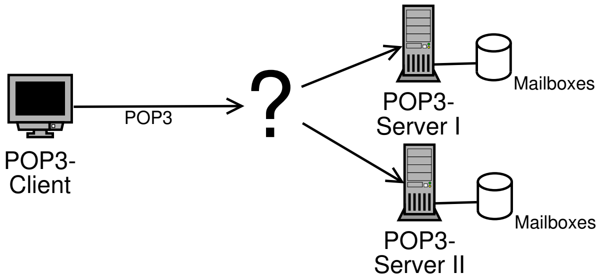
In a small mail system you have one POP3 server that has stored all mailboxes locally on an attached disk. All clients directly connect to this server to access the mail.

Figure 1-1. One POP3 server



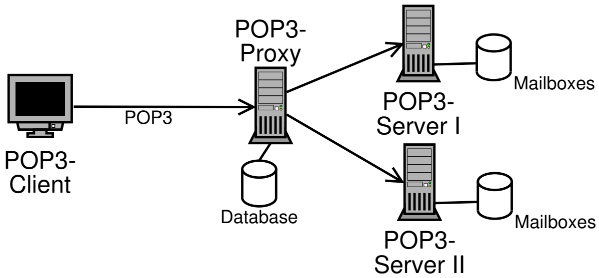
This modell works as long as one server can cope with the load. If you have too many mailboxes or too many accesses per mailbox, you want to split the mailboxes over several servers. But now the client has to know to which server to connect.

Figure 1-2. More POP3 servers



Telling every client about the new server is probably not feasable every time you add new servers. One possible solution is to put all mailboxes on a NFS server and allow access to all mailboxes from all servers. But this will create a new bottleneck: the NFS server. But there is another solution: Put a proxy server between the client and the server.

Figure 1-3. Using a POP3 proxy



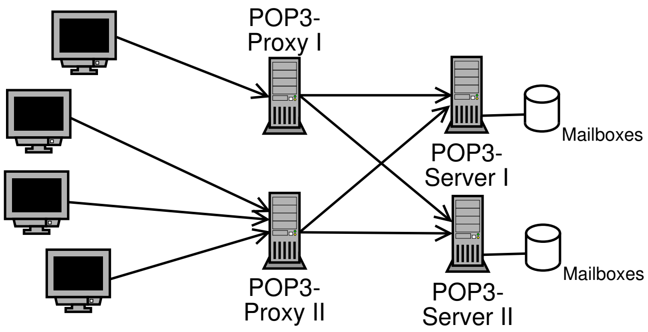
The proxy server will accept the connection from the client, get the username and password from the client, authenticate the user and find out on which backend server this users mailbox is located.

For authentication and the mapping of username to server, the proxy needs some kind of database. This database can either be on the proxy server itself or somewhere else on the net.

After the client is authenticated, the proxy will connect to the proper backend server and blindly forward data back and forth between the client and server until one of them closes the connection.

At that point the proxy might develop to be the bottleneck, but there is no reason not to have several proxies. Through DNS round robin or a load balancing server between the client and the proxy, the load will be distributed over all proxies.

Figure 1-4. The final picture



The *POPular* system implements this architecture including the POP proxy and storage server software.

1.2. The sequence of events in a POP session

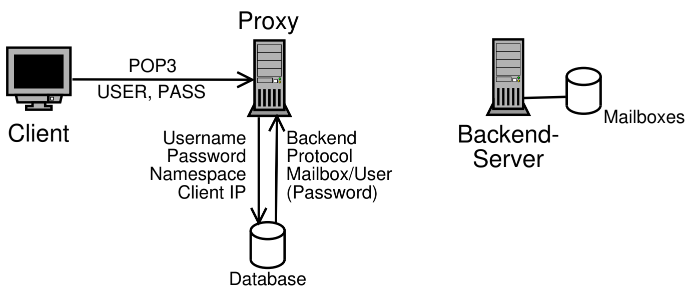
This section explains the several steps in a POP3 session as they are implemented in the *POPular* system.

When a new connection is opened from a client, the POP3 proxy ("**pproxy**") forks and sends a welcome message to the client. The client sends the username and password. Username, password, the client IP and some other information is then checked against one or more databases.

To be as flexible as posssible, these database checks are performed in so called *POPular* database modules (PDM) implemented as shared libraries loaded at runtime into the **pproxy** process. Several databases can be checked after another until one finds and authenticates the user.

The proxy supports IP based virtual servers. It can listen to more than one IP address and port. Depending on which IP address and port a new connection is coming in on, the proxy can use different namespaces for authentication.

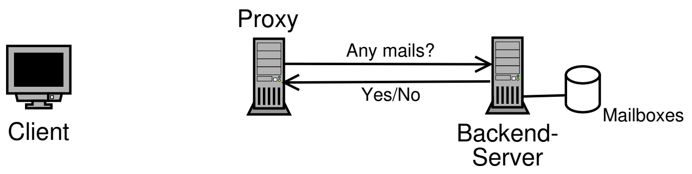
Figure 1-5. New connection and authentication



From the database modules, the proxy will get back the name of the backend server, where the mailbox of this user is stored, and the protocol, which is to be used to connect to this backend. Depending on the protocol either the name of the mailbox or a username and password are also returned.

If the backend server is a *POPular* server, the proxy now sends a mail check requests to the backend server. This is a small UDP datagram only containing the name of the mailbox. The **pcheckd** process on the server will receive this mailbox name and check whether there are any mails in this mailbox It will tell the proxy by sending a UDP datagram back. Statistics have shown that about 80% of all accesses to a mailbox will find an empty mailbox, so it makes sense to check for this and optimize the common case. See below for what happens if the mailbox is empty.

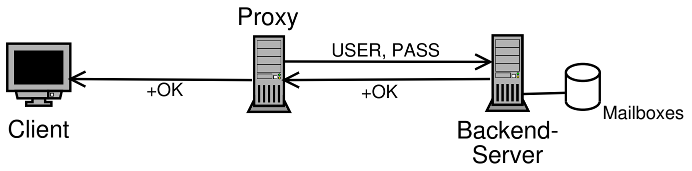
Figure 1-6. Checking mailbox



If there are any mails in the mailbox the proxy will now open a TCP connection to the storage server. Depending on the procotol to be used, two different things can happen:

If the protocol is POP3 indicating a normal POP3 server is used as backend, the username and password that the proxy got from the PDM modules is send to the server. (This username and password might well be the same as the one the client sent, but this is not required.) The backend server will then authenticate the user again and answer with a positive or negative answer, which is relayed back to the client.

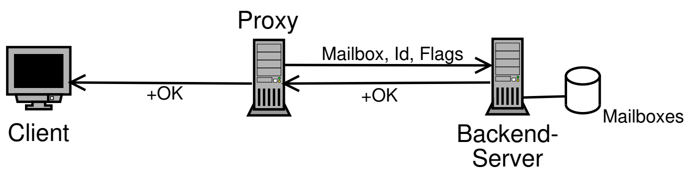
Figure 1-7. Connection to a POP3 server



Because the user is already authenticated it doesn’t really make sense to send the username and password and check them again on the backend server. That is why the *POPular* package contains a special server ("**pserv**") that will not check the user again, but just use a mailbox name supplied by the proxy.

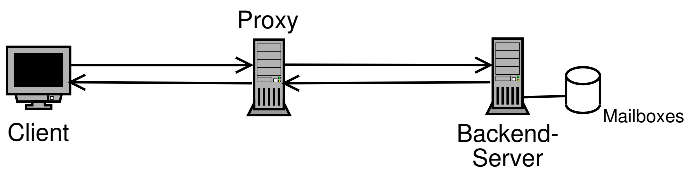
If the protocol is XPOP indicating that a *POPular* backend server is used, the proxy first sends the name of the mailbox in a line by itself, then an ID which is used for all logging messages to easily find corresponding entries in the proxy and server log and then a line with flags. The POP server will answer with "+OK", which is relayed back to the client.

Figure 1-8. Connection to a *POPular* server



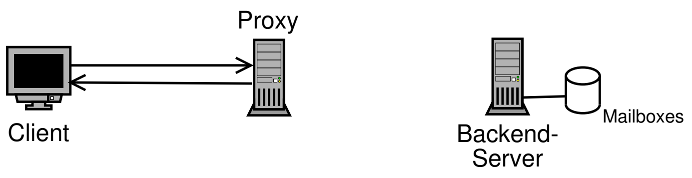
From now on the client and server will exchange POP3 requests and responses. The proxy ferries the data back and forth without interfering until one or the other closes the connection.

Figure 1-9. Proxying between client and backend server



If the mailbox is empty the proxy server will handle this POP connection alone, i.e. it will answer all requests with "no mail" or whatever is appropriate for an empty mailbox. This reduces the load on the storage servers considerably.

Figure 1-10. Empty mailbox



Chapter 2. Installation of the *POPular* system

2.1. Packages

There might be precompiled *POPular* packages available for your system. See the *POPular* web page (http://www.remote.org/jochen/mail/popular/).

If you are installing *POPular* with a package manager like **rpm** or **dpkg** you can probably ignore most information in the rest of this chapter.

2.2. Compilation

Install *POPular* by unpacking the distribution, running **./configure**, **make** and **make install**. This will compile all programs and install them in `/usr/local/bin` and `/usr/local/sbin`. Some *POPular* database modules will be compiled and installed into `/usr/local/lib/popular`. Also, all man pages are made and installed into `/usr/local/man`. Call **configure** with the option `--prefix=PREFIX` if you want to install somewhere else.

The utility scripts, configuration files, etc. are not automatically installed.

The main documentation is created from a SGML/XML source file using the DocBook standard. As there are quite a lot of strange programs and files needed to get the DocBook stuff running, the documentation is provided in the distribution in the compiled HTML, Postscript and RTF version. Look in the `doc` directory for the files named `popular.html`, `popular-htmldoc.tar.gz`, `popular.ps` and `popular.rtf`. These files are not installed automatically. If you want to make the documentation yourself, use the `--enable-docbook` option to **configure**.

Along with the normal GNU **configure** options, the following options can be used:

--with-confdir=DIR

Use DIR as configuration directory instead of `/etc/popular`.

--with-pdm=LIST

List the PDM modules to build. LIST must be a space separated list of module names. See the chapter about PDM modules for a list of modules. By default all available modules are compiled.

Some modules will need external libraries not available on all systems. If the **configure** script can't find those libraries, it will remove these modules from the list and tell you about it.

--enable-test

Enable compilation of test suite in the 'test' directory. After compilation 'make test' can be used to run the tests. This is currently very incomplete.

--with-cdb=DIR

Name the directory, where the source code of Dan Bernstein's cdb library can be found. This is needed, if you want to compile the cdb PDM.

--disable-readline

Disable readline support for the **pcontrol** utility.

--enable-docbook

Enable compiling the main documentation from the SGML/XML Docbook sources into HTML files etc. To do this you need the Docbook tools installed. If your style sheets are installed at some strange location, you can add the directory as an argument to this options. Otherwise the configure script will try to guess, where they are.

--enable-ssl

Enable SSL/TLS support using OpenSSL. SSL/TLS support is currently in beta state. If your OpenSSL library is installed at some strange location, you can add the directory as an argument to this option. Otherwise the configure script will try some default locations.

POPular is being developed on Linux systems and currently only tested on Linux. It should mostly work on any POSIX system, but there are some things (like shared mmaped files and file descriptor passing), which might not work on other systems. Currently it mostly compiles on Solaris, but is not tested very much. If you can help porting *POPular* to other systems, you are most certainly welcome.

2.3. Common setup for proxy and storage server

First you have to decide which user and group to use for the *POPular* programs. If you are (for testing purposes) running the proxy and the backend server on the same machine, both have to use the same user and group, otherwise you can use different users and groups though it is not recommended. Choose a user and group that are not used for anything else on the system. Don't use 'root'. All mailbox directories and mail message files must belong to the user and group you choose. This manual will use the user and group name 'pop'.

The following script can be used to set up the default directories used by *POPular*. If you want to use any other directories, change the script and call the programs with the proper options.

```
#!/bin/sh

POPULAR_USER=pop
POPULAR_GROUP=pop

mkdir -p             /etc/popular
chmod 0755           /etc/popular

mkdir -p             /etc/popular/capa
chmod 0755           /etc/popular/capa

mkdir -p             /var/log/popular
chown $POPULAR_USER  /var/log/popular
chgrp $POPULAR_GROUP /var/log/popular
```

```
chmod 0755           /var/log/popular

mkdir -p             /var/run/popular
chown $POPULAR_USER  /var/run/popular
chgrp $POPULAR_GROUP /var/run/popular
chmod 0755           /var/run/popular
```

2.4. Setting up the proxy server

On the proxy server two program (**pproxy** and **ringd**) will be running.

You should have a file named `pproxy.rc` in the `/etc/popular` directory. Sample files can be found in the `conf` directory of the *POPular* distribution. Make sure `pproxy.rc` is executable.

First start **ringd** as user 'root'. **ringd** will run in the background and bind TCP ports for **pproxy** because **pproxy** runs as a normal user.

Next start **pproxy** as the user and group you chose. You should see some log messages appearing in `/var/log/popular/pproxy`.

To configure the running **pproxy** use the **pcontrol** program. Always start **pcontrol** as the user you have chosen for the **pproxy**. If everything is set up properly you can just call `/etc/popular/pproxy.rc` to do the first configuration.

2.5. Setting up the storage server

You should have a file named `pserv.rc` in the `/etc/popular` directory and it should be executable for the user you have chosen to run the server as. A sample file can be found in the `conf` directory of the *POPular* distribution.

Decide where to put the mailboxes, the sample config file uses the directory `/pop` for this. All mailbox directories must be readable and writeable by the user you chose for running **pserv**.

Start the server as the user and group you chose: **pserv**. You should see some log messages appearing in `/var/log/popular/pserv`.

To configure the running **pserv** use the **pcontrol** program. Always start **pcontrol** as the user you have chosen for the **pserv**. If everything is set up properly you can just call `/etc/popular/pserv.rc` to do the first configuration.

If you want to use the **pcheckd** daemon, start it as the same user you used for **pserv**. You need at least to use the option -m to tell **pcheckd** where the mailbox directory is (`/pop` by default).

Chapter 3. Configuring pproxy

The configuration of the *POPular* proxy is done by sending commands to the running proxy server via the **pcontrol** utility program. When the proxy server is started it will configure itself with a few default values but will not open any ports. Much like a UNIX kernel, that has no network interfaces configured after booting and only a minimal root filesystem mounted, the proxy then has to be contacted to be configured for use.

The **pcontrol** utility program will read commands for the running **pproxy** from the command line, a file or from STDIN.

On startup the init-Skript will read the commands in the file `/etc/popular/pproxy.rc` and execute them.

3.1. Command line options

3.2. Runtime configuration

The configuration of the *POPular* proxy is done by sending commands to the running server via a UNIX domain socket. This socket is in the run directory (default: `/var/run/popular`) and named `pproxy.PID.ctrl`. PID, in this case, is the process id of the parent process. For easy access, there is a link from `pproxy.ctrl` to the real socket file.

The **pcontrol** utility program is provided to send commands to this socket. See the man page for details.

Normally, after **pproxy** has started, the file `/etc/popular/pproxy.rc` should be used to set the start configuration of the server. There is an example of this file in the `conf` directory of the distribution. The supplied RedHat and Debian init scripts will use this file to do the initial configuration.

3.2.1. The show and set commands

The following read-only variables are recognized. The content of these variables can be read by sending the command **show VAR** to the proxy.

Each variable has one of the following types: `Bool`, `Int`, `File`, `Dir`, `String`, `Id`, or `Time`. `Bool` variables can be set to 0=false=no=off or 1=true=yes=on. Directories named in variables of type `Dir` must exist. Variables of type `Id` hold strings which can only contain chars, numbers, the underscore (_), the hyphen (-), and the dot (.). `Time` variables hold time intervals. They can be set using something like ’5d3h10m4s’.

`id`	Id	Id (name) of server. This is always "pproxy".
`pid`	Int	The process id of the **pproxy** parent process.
`rundir`	Dir	This is the directory set by the `--rundir` option.
`sessionlimit`	Int	The maximum compiled in value for maxsession.
`version`	String	The version of the **pproxy** program.

The following read-write variables are recognized:

`allowsslv2`	Boolean	Allow SSLv2 connections. Only used when SSL/TLS is compiled in. Default is off. SSLv2 is considered less secure than SSLv3 and TLS, but some clients might need it. Changes in this variable will only be applicable for virtual servers that are configured after the change!
`authtimeout`	Time (0-1h)	This is the timeout while in authorization state. 0 means no timeout.
`backlog`	Int (5-1024)	Backlog parameter for the listen(2) system call. When this is changed future listen(2) system calls will get the new parameters. Already listening sockets are not affected. To activate the new setting the virtual server has to be taken offline and online again.
`capadir`	Dir	This is the directory where the definitions for capability lists are to be found. Consult the chapter about capabilities for details.
`checkport`	Int (1-65535)	UDP Port, that the **pcheckd** on the backend server listens on. All children spawned after this is changed will use the new value.
`checktimeout`	Time (1s-1m)	Timeout in seconds after which a UDP packet sent to the **pcheckd** server is given up as lost. All children spawned after this is changed will use the new value.
`defaultns`	Id	This is the default namespace used, if the namespace is USER and there is no namespace in the username sent from the client. All children spawned after this is changed will use the new value.
`fallback`	Id	If the username is not found in the local database, the proxy server can connect to a fallback POP3 backend server and try authenticating the user there. This is the name of the backend as defined with the **backend** command. All children spawned after this is changed will use the new value.
`idletimeout`	Time (10m-1d)	Timeout in seconds when a session is idle. If no data is read after this timeout the connection will be closed. All children spawned after this is changed will use the new value. This is what RFC1939 calles the 'inactivity autologout timer'. RFC1939 doesn't allow this to be set smaller than 10 minutes, so *POPular* doesn't allow this. See also `proxytimeout`.
`logfile`	File	Name of the logfile. This will take effect the next time the log file is reopened.
`maxlocalload`	Id (0-9999)	If the system load average for the last minute is equal to this value or higher, no new connections are accepted. This is an integer value between 0 and 9999. 0 means that the load is not checked. When the server notices that the load is too high (or ok after beeing to high), this event is logged. The load average is not a really good measure of the system load so this is more of a last ditch effort to keep the system usable than a good way of controlling the load. You probably have to experiment with different values of maxsession and maxlocalload to find the best values for your system. Note that the load check currently only works on systems with `/proc/loadavg`.
`maxsession`	Id	Maximal number of proxy child processes. This means that not more then this number of simultaneous connections can be handled. You probably want to set this large enough to handle the biggest estimated load on your system without running out of ressources. Note that there is a maximum value for this compiled into the binary. The value of this variable can currently only be increased, not decreased.

`pdmdir`	Dir	Directory where *POPular* database modules (PDM) are stored.
`proxytimeout`	Time (10m-1d)	Timeout in seconds when proxying data. If no data is read from the client or backend for this time the connection will be closed. All children spawned after this is changed will use the new value. This is what RFC1939 calles the 'inactivity autologout timer'. RFC1939 doesn't allow this to be set smaller than 10 minutes, so *POPular* doesn't allow this. See also `idletimeout`.
`sessiontimeout`	Time (0s-1d)	Overall timeout for the length of a session in minutes. Regardless of what is happening, a session will be killed after `sessiontimeout` minutes. This value must be between 0 and 1 day. If `sessiontimeout` is 0, no limit is enforced. Note that the time real session time may be somewhat longer, because the authentication phase is not included in this timer. There is a separate timeout for this: `authtimeout`.
`sidprefix`	Id	This is the prefix for all session IDs. Default is the hostname (without the domain). If this is set to a different value on every proxy server in a cluster, session IDs will be unique over the whole cluster. The session ID will be the concatenation of this prefix, the UNIX timestamp (seconds since the epoch 1970-01-01 00:00:00), and the process ID of the pproxy process handling this session. These three components are separated by dots. Example: "p02.990738012.7845".
`tlsdir`	Dir	In this directory all the key and signature files for SSL/TLS are to be found.

In all places where host names or IP numbers must be given, either one can be given. A host name is resolved to an IP number at the moment of configuration. The IP number is then saved internally. So if you change something in your network and IP numbers change, you have to redo the configuration.

3.2.2. The shutdown command

There are four variants of the **shutdown** command:

shutdown parent will shutdown the parent process, but leave the children running.

shutdown children asks the parent to kill all its children, but keep running itself.

shutdown all asks the parent to kill all its children and then exit itself.

With **shutdown delayed**, the parent will close all listening sockets, i.e. it will not accept any new connections. It will then wait for all its children to die and then exit itself.

Several signals can be sent to the server to achieve the same effect. See below.

3.2.3. The capa command

The **capa** command is used to manage user defined capability sets. The following subcommands are supported:

list	List IDs of all loaded capability sets.
load `NAME`	Load a set of capabilities from disk. NAME is the file name of this list in the `capadir` directory and the ID that will be used in **pproxy**.
del `NAME`	Delete a set of capabilities from **pproxy** memory.

show `NAME`	Show the capabilities in a capability set.

Note that the **capa** command for **pserv** uses a different syntax!

3.2.4. The debug command

The **debug** can be used to enable or disable certain debug logging options. If **debug** is used without arguments, the current debug options are printed. If the **debug** command is followed by one of '=', '+', or '-' the debug options following after this character are set, added to the current set or deleted from the current set, respectively.

The following debug logging options are recognized:

`AUTH`	Authentication details (but without passwords).
`RINGD`	Communication with the ringd process.
`CTRL`	Control requests and answers.
`IO`	Low level IO functions.
`MAIN`	General messages about what the program is doing.
`NET`	Network stuff...
`PASS`	Passwords are logged. This might be a security risk if somebody can access your logfiles, so this should normally not be used. Note that ALL includes this.
`POP`	POP3 dialogs.
`TLS`	SSL/TLS messages if compiled in.
`ALL`	Pseudo option including all of the above.
`NONE`	Pseudo option meaning 'no option'.

3.2.5. The vserv command

The *POPular* proxy features virtual server support. Depending on which IP number and port a request comes in, it can use different namespaces for authenticating a user. Currently there is a maximum number of 32 virtual servers compiled in. Virtual servers are configured with the **vserv**. Each virtual server has an ID which is used to address it with the configuration commands. The ID can be freely chosen, but only letters, digits, underscore, hyphens and dots are allowed. The following subcommands are recognized:

vserv list

Show a list of all configured virtual servers. Only the ID of the virtual server is shown. It can be used to show or set details of its configurations with other subcommands.

vserv flush

Deletes all virtual servers. All listening ports are closed.

vserv show VSERV

Show details of the configuration of the virtual server VSERV. The configuration is output in the format of the **vserv conf** command, so it can be easily cut-and-pasted. See below for the configuration options.

vserv conf VSERV [PARAMETER ARG] ...

Set configuration parameters for the virtual server VSERV. If VSERV doesn't exist it will be created. One or more parameters can be set in one command. Parameters that are not set are unchanged or get a default value for new servers. See below for a list of parameters.

vserv del VSERV

Delete the virtual server VSERV. The corresponding file descriptor is closed.

The following parameters can be set for virtual servers:

`iface`	Interface this virtual server should listen on. If this is the string "ANY", this virtual server will be bound to IPADDR_ANY. Otherwise this must be an IP number or a host name. A host name will be resolved to the first IP number for this hostname at the moment the command is given. Obviously the IP number should belong to the host the **pproxy** command is running on. The IP number can only be changed if the virtual server is currently in the state "offline".
`port`	This is the TCP port, this virtual server should listen on. It defaults to the well-known port for the protocol given as the **prot** parameter. The port can only be changed if the virtual server is currently in the state "offline".
`prot`	This is the protocol that should be spoken on this virtual server. Currently POP3 and POP3s is supported. The default is POP3. See the chapter on SSL/TLS for details. This is only available if the server was configured with SSL/TLS support.
`starttls`	Set this to "off" to disable the STARTTLS command. Set this to "optional" to allow the STARTTLS command. Set this to "force" to force use of the STARTTLS command before authentication. See the chapter on SSL/TLS for details. This is only available if the server was configured with SSL/TLS support.
`capa`	This is the capability list that should be used on this virtual server. See the chapter about capabilities for details.
`state`	Each virtual server can be in one of four states. See below for a list of states.
`namespace`	This is the "namespace", that should be used for all authentications from users coming in on this virtual server. If the special name space "USER" is set here, the real name space will be set with the POP3 USER command. This is sometimes useful for access by the postmaster or support staff. Instead of sending the user name after the USER command, the client has to send the user name, an equal sign ('=') and the namespace after the USER command. See the Authentication chapter for more information about namespaces.
`bannerok`	This is the POP3 greeting banner used when the virtual server is online. A "+OK" is output before the banner. The banner should be something like "host.doma.in POP3 server ready".
`bannererr`	This is the POP3 greeting banner used when the virtual server is disabled. A "-ERR" is output before the banner. The banner should be something like "host.doma.in POP3 server currently disabled".

Most parameters can be changed at any time and will take effect immediately for any new connections. Old connections are never affected by any change in parameter.

A virtual server can be in one of the following states:

`offline`	This virtual server is configured, but not in use. There is no file descriptor listening to the port. To change the IP number of port the virtual server has to be taken offline.
`disabled`	The virtual server is listening on its port, but all connections will be greeted with the "bannererr" banner and immediately disconnected.
`fake`	The virtual server is listening on its port, but all users will just see an empty mailbox. The client is asked for username and password, but it is NOT properly authenticated. The reason is that the virtual server might has been disabled by the admin, because the authentication is not working properly at the moment.
`online`	The virtual server is online and working.

3.2.6. The backend command

The *POPular* proxy server can connect to several backend servers Currently there is a maximum number of 32 backends compiled in. Backend servers are configured with the **backend**. Each backend server has an ID which is used to address it with the configuration commands. The ID can be freely chosen, but only letters, digits, underscore, hyphens and dots are allowed. The following subcommands are recognized:

backend list

Show a list of all configured backends. Only the ID of the backends is shown. It can be used to show or set details of its configurations with other subcommands.

backend flush

Deletes configuration of all backends.

backend show BACKEND

Show details of the configuration of the backend BACKEND. The configuration is output in the format of the **backend conf** command, so it can be easily cut-and-pasted. See below for the configuration options.

backend conf BACKEND [PARAMETER ARG] ...

Set configuration parameters for the backend BACKEND. If BACKEND doesn't exist it will be created. One or more parameters can be set in one command. Parameters that are not set are unchanged or get a default value for new servers. See below for a list of parameters.

backend del BACKEND

Delete the backend BACKEND.

The following parameters can be set for all backends:

`host`	Hostname or IP number of this backend. A host name will be resolved to the first IP number for this hostname at the moment the command is given.
`port`	This is the TCP port, where the backend server should listens on. It defaults to the well-known port for the protocol given as the **prot** parameter.
`prot`	This is the protocol that should be spoken with this backend. Currently only POP3, XPOP and CXPOP are supported. The default value is POP3.
`state`	Each virtual server can be in one of three states. See below for a list of states.

Most parameters can be changed at any time and will take effect immediately for any new connections. Old connections are never affected by any change in the parameters.

A backend can be in one of the following states:

`offline`	This backend is configured, but not in use. All connections that would result in a use of this backend will be closed after an error message is sent.
`fake`	This backend is configured, but not in use. The proxy server will fake an empty mailbox.
`online`	The backend server is online and working.

3.2.7. The pdm command

The **pdm** ist used to configure the authentication modules. The following sub commands are available:

pdm list

Show list of modules. This list is ordered. Modules will be called upon in order to find and authenticate an user.

pdm flush

Delete all modules.

pdm add ID MOD ARG ...

Add a module to the list of modules. ID is a unique ID of this module. MOD is the name of the module. The module will be looked for in the directory set in the config variable 'pdmdir'. A 'libpdm_' will be prepended before the module name and a '.so' appended. All arguments (ARG ...) will be handed over to the module.

pdm del ID

Delete this module from the list.

pdm reload ID

Reload module. The action depends on the module. Generally it will mean that files are reopend etc.

pdm show ID

Show module name and arguments with which this module was loaded.

3.2.8. The prng command

The **prng** ist used to seed the pseudo random number generator. It is only used when SSL/TLS is compiled in.

pdm FILENAME [BYTES]

Read BYTES bytes (default 1024) from file FILENAME. Normally the file will be the `/dev/random` for maximum security or `/dev/urandom` for somewhat less security. Reading from `/dev/random` might block if there is not enough entropy available. You can also read from a normal file if you have some random data in there. Make sure you understand what this is all about, otherwise your precious SSL/TLS server might not be so secure after all. Use this command before you issue any **vserv** commands, so that the PRNG is initialized before use.

3.3. Signals

The **pproxy** parent understands the following signals:

`SIGHUP`	Reopen log file
`SIGTERM`	The parent process will kill all its children and then shutdown itself cleanly. This is the same as sending the **shutdown all** command to the server.
`SIGQUIT`	The parent process will shutdown cleanly, but the children are left running. This is the same as sending the **shutdown parent** command to the server.
`SIGINT`	The parent will kill all its children and keep running itself. This is the same as sending the **shutdown children** command to the server.
`SIGPWR`	The parent will close its listening sockets (i.e. stop accepting new connections) and then wait for all children to exit before shutting itself down cleanly. This is the same as sending the **shutdown delayed** command to the server.

The child processes have no special signal handling, i.e. they will die on receiving `SIGTERM`, `SIGQUIT`, `SIGINT`, or `SIGHUP`.

Chapter 4. Configuring pserv

4.1. Starting pserv

pserv is always started as the 'pop' user (or whatever user you decided should be used). Normally there are no command line options required.

The `--logfile` option can be used to name the initial log file (default is `/var/log/popular/pserv`). The name of the logfile can later be changed with the **set logfile** directive.

The `--rundir` option can be used to change the run directory, where some files needed for the operation of **pserv** are stored. This can't be changed while the program is running.

All the rest of the configuration is done while the server is running. A server that has just started doesn't listen to any ports and has only default values set for its configuration variables.

4.2. Runtime configuration

The configuration of the *POPular* server is done by sending commands to the running server via a UNIX domain socket. This socket is in the run directory (default: `/var/run/popular`) and named `pserv.PID.ctrl`. PID, in this case, is the process id of the parent process. For easy access, there is a link from `pserv.ctrl` to the real socket file.

The **pcontrol** utility program is provided to send commands to this socket. See the man page for details.

Normally, after **pserv** has started, the file `/etc/popular/pserv.rc` should be used to set the start configuration of the server. There is an example of this file in the `conf` directory of the distribution. The supplied RedHat and Debian init scripts will use this file to do the initial configuration.

4.2.1. The show and set commands

The following read-only variables are recognized by **pserv**. The content of these variables can be read by sending the command **show VAR** to the server. Calling **show** without argument will output a list of all configuration variables.

Each variable has one of the following types: `Bool`, `Int`, `File`, `Dir`, `String`, `Id`, or `Time`. `Bool` variables can be set to 0=false=no=off or 1=true=yes=on. Directories named in variables of type `Dir` must exist. Variables of type `Id` hold strings which can only contain chars, numbers, the underscore (_), the hyphen (-), and the dot (.). `Time` variables hold time intervals. They can be set using something like '5d3h10m4s'.

`id`	Id	Id (name) of server. This is always "pserv".
`pid`	Int	The process id of the **pserv** parent process.
`rundir`	Dir	This is the directory set by the `--rundir` option.
`sessionlimit`	Int	The maximum compiled in value for maxsession.
`version`	String	The version of the **pserv** program.

The following read-write variables are recognized by **pserv**. The content of these variables can be read by sending the command **show VAR** to the server. They can be set by sending the command **set VAR CONTENT**.

`backlog`	Int (5-1024)	Backlog parameter for the listen() system call. The server has to be taken offline and online again for this value to take effect.
`capadir`	Dir	This is the directory where the definitions for capability lists are to be found. Consult the chapter about capabilities for details.
`localip`	Str	This is the IP number the server should bind to. It can be "ANY", in which case the server will bind to `INADDR_ANY`. A change will take effect when the next **server online** is given.
`logeachmsg`	Bool	Can be 0 (off) or 1 (on). If this is on, the server will log deleted and read mail. This is an experimental feature and the format of this logs etc. might change.
`logfile`	File	Name of the logfile. This will take effect the next time the log file is reopened. The default value for this is the parameter given to the `--logfile` option or, if no log file was specified on the command line, `/var/log/popular/pserv<`.
`maxlocalload`	Int (0-9999)	If the system load average for the last minute is equal to this value or higher, no new connections are accepted. This is an integer value between 0 and 9999. 0 means that the load is not checked. When the server notices that the load is too high (or ok after beeing to high), this event is logged. The load average is not a really good measure of the system load so this is more of a last ditch effort to keep the system usable than a good way of controlling the load. You probably have to experiment with different values of `maxsession` and `maxlocalload` to find the best values for your system. Note that the load check currently only works on systems with `/proc/loadavg`.
`maxsession`	Int	Maximal number of child processes. This means that not more then this number of simultaneous connections can be handled. You probably want to set this large enough to handle the biggest estimated load on your system without running out of ressources. Note that there is a maximum value for this compiled into the binary. The value of this variable can currently only be increased, not decreased. When `maxsession` is reached, a file called `pserv.maxsession` in the rundir (`/var/run/popular`) is created. If the session count decreases below `maxsession`-1, this file is deleted. **pcheckd** uses this file to inform the proxy that the maximum number of sessions is reached.
`popdir`	Dir	Directory where all the mailboxes are.
`idletimeout`	Time (10m-1d)	Timeout in seconds when a session is idle. If no data is read after this timeout the connection will be closed. All children spawned after this is changed will use the new value. This is what RFC1939 calles the 'inactivity autologout timer'. RFC1939 doesn't allow this to be set smaller than 10 minutes, so *POPular* doesn't allow this.
`sessiontimeout`	Time (0s-1d)	Overall timeout for the length of a session in minutes. Regardless of what is happening, a session will be killed after `sessiontimeout` minutes. This value must be between 0 and 60*24 (1 day). If `sessiontimeout` is 0, no limit is enforced.
`servport`	Int (1-65535)	TCP Port for the XPOP protocol. The server will listen on this port.

`statusheader`	Bool	If this is set to true, a header line 'Status: RO' will be included in all mail that has already been read, i.e. that is found in the 'cur' directory. This line is not in the file on disk, the file on disk never changes, it is only included while sending out the mail to the client. This if off by default.

4.2.2. The server command

To start and stop the server the **server** command is used. With **server online** the server will start listening on its TCP port. With **server offline** the server will close the listening socket. Make sure to set the right port number with **set servport PORT** before going online. With **server status** the current status of the server can be seen.

4.2.3. The shutdown command

There are four variants of the **shutdown** command:

shutdown parent will shutdown the parent process, but leave the children running.

shutdown children asks the parent to kill all its children, but keep running itself.

shutdown all asks the parent to kill all its children and then exit itself.

With **shutdown delayed**, the parent will close its listening socket, i.e. it will not accept any new connections. It will then wait for all its children to die and then exit itself.

Several signals can be sent to the server to achieve the same effect. See below.

4.2.4. The capa command

The **capa** is used to set the list of capabilities for this **pserv**. **capa show** will display the current setting. Use **capa set *NAME*** to choose which set of capabilities to use. `NAME` can be one of the following:

`ERROR`	The POP3 **CAPA** command will be answered with an error message simulating an old server without support for the **CAPA** command.
`NONE`	The capability list is empty.
`DEFAULT`	A default list of capabilities is used. This describes all the capabilities implemented by *POPular* and is consistent with the capability set of the **pproxy** program if it was configured with **vserv conf VS capa DEFAULT**.
`everything else`	Must be a file name in the `capadir` directory. The contents of this file are aused as the list of capabilites.

Note that the **capa** command for **pproxy** uses a different syntax!

4.2.5. The debug command

The **debug** can be used to enable or disable certain debug logging options. If **debug** is used without arguments, the current debug options are printed. If the **debug** command is followed by one of '=', '+', or '-' the debug options following after this character are set, added to the current set or deleted from the current set, respectively.

The following debug logging options are recognized:

`CTRL`	Control requests and answers.
`IO`	Low level IO functions.
`MAIN`	General messages about what the program is doing.
`NET`	Network stuff...
`POP`	POP3 dialogs.
`ALL`	Pseudo option including all of the above.
`NONE`	Pseudo option meaning 'no option'.

4.3. Signals

The **pserv** parent understands the following signals:

`SIGHUP`	Reopen log file
`SIGTERM`	The parent process will kill all its children and then shutdown itself cleanly. This is the same as sending the **shutdown all** command to the server.
`SIGQUIT`	The parent process will shutdown cleanly, but the children are left running. This is the same as sending the **shutdown parent** command to the server.
`SIGINT`	The parent will kill all its children and keep running itself. This is the same as sending the **shutdown children** command to the server.
`SIGPWR`	The parent will close its listening sockets (i.e. stop accepting new connections) and then wait for all children to exit before shutting itself down cleanly. This is the same as sending the **shutdown delayed** command to the server.

The child processes have no special signal handling, i.e. they will die on receiving `SIGTERM`, `SIGQUIT`, `SIGINT`, or `SIGHUP`.

Chapter 5. Configuring pcheckd

5.1. Command line options

5.2. Signals

pcheckd understands the following signals:

`SIGHUP`	Reopen log file
`SIGTERM`	Shut down **pcheckd** cleanly.
`SIGQUIT`	
`SIGINT`	
`SIGPWR`	
`SIGUSR1`	Disable debug logging
`SIGUSR2`	Enable debug logging

Chapter 6. POPular Database Modules (PDM)

6.1. Introduction

POPular uses a very flexible database lookup and authentication method based on shared libraries loaded at runtime. These libraries are called "POPular Database Modules (PDM)". Several modules are currently provided with *POPular*. More modules can be written easily as the API is quite simple.

After a client program sends the username and password all loaded modules will be called in order and given the clients IP number, the used protocol, the used namespace, and the username and password. Each module can check, whether it can identify and authenticate the user. If a module authenticates the user, no more modules are called. If the module can identify the user, but doesn't allow access for any reason (like a wrong password), all modules are again called and asked whether they can authenticate the user.

Database modules can be loaded into and unloaded from **pproxy** at runtime. See the description of the **pproxy** configuration for details.

For testing database modules alone and together with other modules, the **ptestpdm** program is provided. For details see the documentation for this command.

6.2. Any module (libpdm_any.so)

This is a test module which will accept any user. If you are new to all this, this is a good place to start.

6.3. Master password module (libpdm_master.so)

Using a master authentication file for access to all mailboxes with special master password. The name of the file is given as an argument when initializing the module (default: `/etc/popular/masterauth.conf`). This is a plain text file. Each line contains an IP number in dotted quad notation, a colon, a namespace name, another colon and a crypted password. If the IP number matches the IP number of the POP3 client, the namespace name matches the accessed namespace, and the password matches the given POP3 password, access is granted. Empty lines and lines beginning with a '#' symbol are ignored.

This feature is intended to be used by the postmaster or support staff. Connections authenticated with the master password are flagged. New mail read is not marked as read, so that the customers MUA will not get confused. Mail that is deleted, WILL REALLY BE DELETED.

6.4. Berkeley DB Version 2 module (libpdm_db2.so)

Using Berkeley DB Version 2 hash files. The hash file used is named after the namespace and is looked for in the directory that was given as argument when initializing the module. (default: `/etc/popular/db/`). An example: `/etc/popular/db/test.db` is the file for the namespace 'test'.

The lookup key in the DB file is the POP3 username. There is no trailing zero after the key or data. The data should be a colon separated list of the crypted password, the ID of the backend server and the mailbox name. A Perl script **db-create.pl** is supplied (along with example files) in the `auth` subdirectory to create this DB file from a text file containing lines with a colon separated list of username, crypted password, backend server name and mailbox. The 'source' file can contain empty lines and comment lines beginning with a '#' symbol.

6.5. Berkeley DB Version 3 module (libpdm_db3.so)

Using Berkeley DB Version 3. Apart from that its the same as db2.

6.6. CDB (constant data base) module (libpdm_cdb.so)

Using Dan Bernstein's CDB.

Chapter 7. Logfiles

All *POPular* programs log directly into files. `syslog` is not used. The default log directory is `/var/log/popular`. This directory should be owned by the user running all the commands and only writable by this user. The commands **pproxy**, **pserv**, **pcheckd**, **psmtpd**, and **pdeliver** write to a logfile named after the command.

7.1. Logfile format

The log file format is carefully designed to be easily **grep**pable in shell or Perl scripts. The date and time stamps use the compact ISO 8601 format, which is concise, unambiguous and easily sortable. See http://www.cl.cam.ac.uk/~mgk25/iso-time.html for details.

Each log entry is in one line and contains the following fields separated by a space character:

Date	In the ISO format YYYYMMDD.
Time	In the ISO format HHMMSS.
Hostname	The hostname without domain.
Program	The name of the program logging this event.
Process ID	The process ID of the program logging this event.
Session ID	This ID is the same for all log lines belonging to the same session. It is build from the contents of the `sidprefix` config variable, the start time of the session and the process ID of the **pproxy** process handling this session. For the same session, the session ID is the same in both the **pproxy** and the **pserv** logs. If there is no session ID associated with this log line a single hyphen ('-') is printed.
Level	The level of this message. See below for a list of logging levels.
Unique message number	A unique 4-digit hex number for this logging message. A list of all numbers and their corresponding messages and meanings is automatically generated from comments in the source code. These numbers are not reused if they are retired after a program change. For debug messages this number if always '0000'. A list of all message numbers along with a description what they mean is available from the popular-log(7) man page.
Name	This is a description of the event causing this log entry. This is always a single word, if necessary underscores are used. This makes the log messages easily 'grep'able. Debug messages print different information here, see below.
Additional text	Additional text like human readable error messages, usernames, IP addresses, etc. This is the only field in a log line, which can contain spaces. Non-printable chars are escaped. If a message if too long, it will be truncated and '...' added.

Debug messages use a slightly different format: The message number is always '0000' and instead of the message name a string containing the following parts separated by colons (':') are printed: The debug type, the current C function, the C source file and the line number in the source file.

7.2. Log levels

The following log levels are used. They are chosen based on the idea that it should be easy for the system administrator to find out, whether any action on his part is needed.

DBG	debug message	Debugging messages. Generally, these are only interesting for the developer. A system administrator might enable them to help the developer finding bugs.
INF	informational message	Informational messages produced while the program runs its normal course. They are mainly used for statistics and to show what the program has been doing.
ERR	error message	Messages which denote an error in the normal course of the program. These might be messages like 'password mismatch', i.e. errors based on user input. An ERR message is normally not a concern for the system administrator, but it might be for customer support personnel. ERR messages have an extra label, because they can be found more easily that way. A very high number of ERR messages might indicate a system problem in another part of the system, like database corruption or network problems. The ratio of INF to ERR messages is probably a good indication of those kinds of problems.
ADM	admin intervention required	These messages denote things that need attention by the system administrator. They might signal if the system is near an overload condition etc. The system administrator should inspect those messages regularly and act on it. These messages are not really time critical, maybe they are only a warning that new hardware should be installed, or they might warn of data corruption in one users mailbox. Lots of these messages in a short time frame might indicate a critical situation.
SOS	critical event	These messages denote critical system events like system or disk overload, processes crashing, etc. Generally, they demand an immediate intervention by the system administrator, although the system might try to keep running.
BUG	software bug	These messages are produced by internal assertion code and denote system states that are not supposed to happen. This probably means there is a bug in the program or something else happens, that the developer didn't anticipate. These messages should be reported to the developer.

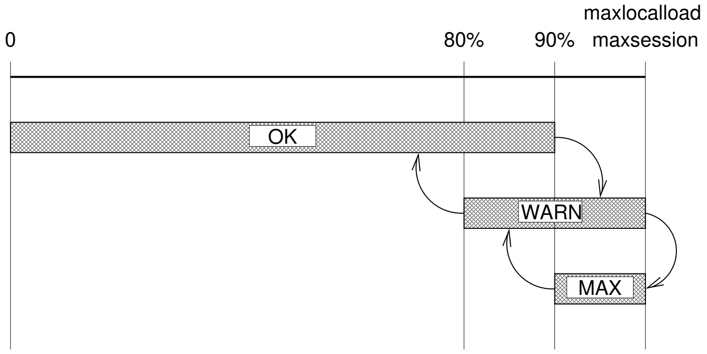
7.3. Reopening log files

A SIGHUP can be sent to the **pproxy**, **pserv**, **pcheckd**, and **psmtpd** commands to reopen the log file. If the new log files can't be opened, the all programs except **psmtpd** will keep logging to the old file. **psmtpd** will not log anything anymore! Note that child processes don't know about this log file change and will keep logging to the old file. This is considered a feature, because all entries from one child will be in the same log file. Children are normally not running for a long time, so this is no problem for log file rotation. Just make sure that after rotating the log file it is not immediately gzipped or copied away and deleted.

7.4. Max session/load warnings

The *POPular* servers will log warnings when certain load and session limits are reached.

Figure 7-1. The three states for logging of load and session limits



The session and load limit can be configured through the `maxsession` and `maxlocalload` config variables. Both are integers.

When 90% of the session or load limit is reached, *POPular* will switch from state 'OK' to state 'WARN', when 100% of the session or load limit is reached, *POPular* will switch to state 'MAX'. When the session count or load falls below 90% in state 'MAX', it will go to state 'WARN'. When the session count or load falls below 80% in state 'WARN', it will go to state 'OK'.

Each change of state is logged. For session state changes a message number 0x0170 ('session_state') is logged, for load state changes a message number 0x0172 ('load_state') is logged. The message contains the name of the new state.

As long as there are session slots available, new connections will be accepted regardless of the session state.

If the load state is 'MAX', no new connections will be accepted until the state drops back to 'WARN'.

Chapter 8. Displaying state of running servers

Both the proxy (**pproxy**) and the server (**pserv**) save their state and some configuration information in a shared memory mapped file. This file can be read at any time to find out about the current connections at their state, some counters and other information.

The state information can be found in the files `/var/run/popular/pproxy.state` and `/var/run/popular/pserv.state`, respectively.

The format of the files is in binary and defined in the C structs shmem_proxy and shmem_server. The first 8 chars contain the magic "POPULAR\0", the second 8 chars the magic "PROXY\0\0\0" or "SERVER\0\0", respectively. Following this is an integer with the version number (currently 1).

The **pstatus** command is supplied for reading both the **pproxy** and **pserv** state files. It will check the magic at the beginning of the file to find the type of file and then dump its contents to stdout. See the documentation for the **pstatus** command for an explanation of the dumped information.

If you want to "freeze" the current status information, just copy the file to a temporary file and redirect the **pstatus** command to read this file.

Note: There is currently no locking on the status file. This will not affect the correct function of the proxy and server, but it might lead to wrong data displayed by the **pstatus** command. This will be fixed at some point.

Chapter 9. Utility programs

There are a few utility programs written in Perl that come in handy in conjunction with the POP server.

9.1. Mail delivery

The **pdeliver** program described elsewhere in this document is used for local delivery of mails into mailboxes. For delivering mails from remote hosts, there is another program, which can be used. `scripts/psmtpd` is a Perl script, which implements a minimal SMTP server. Basically the only thing it can do is deliver a mail to a single mailbox.

There are some variables at the beginning of the file, which need to be defined properly.

> This program is *not* intended to be used as general purpose MTA. It can only be used behind a carefully configured other MTA, that feeds it properly. Never, ever, permit the general Internet access to this server.

9.2. Cleanup

There is a script **pclean** provided, that can be used for all sorts of cleanup chores. At the moment this script is in beta state and not all that well tested.

In the meantime the following three scripts in the `scripts` directory can be used for nightly cleanup chores:

`movedel`

To aid in recovery after accidential deletion of a mailbox, the mailbox is not really deleted but saved under a different name. This script will check the lists of configured mailboxes in `/etc/popular/mb/*` and rename all mailboxes not in one of these files. The new name is simply the old name with the date/timestamp appended.

`expire`

This script expires mails in mailbox after a configurable time. The time can be configured differently for each namespace. Deleted mailboxes have their own expire time. Additionally, this script will remove deleted mailbox directories, when there are no mails left.

`checksize`

This script checks the size of all mailboxes. Mailboxes who are above or near the quota are logged. Additionally a message is put into the mailbox to notify the user about his full or nearly full mailbox.

Chapter 10. Security

10.1. SSL/TLS support

Starting in version 1.5.0 POPular supports SSL and TLS if compiled with the OpenSSL library. You need at least version 0.9.6b. SSL/TLS is only available if POPular was configured with the `--enable-ssl`.

Each virtual server can be configured to either use unencrypted connections or to use SSL/TLS connections or use the RFC2595 style STARTTLS command. Use the **prot** and **starttls** options of the **vserv** command to set this as follows:

1. For usage without SSL/TLS set **prot** to "pop3" and **starttls** to "off".
2. For direct SSL/TLS usage without STARTTLS set **prot** to "pop3s" and **starttls** to "off".
3. For usage of an optionally encrypted connection set **prot** to "pop3" and **starttls** to "optional". The connection will start out unencrypted and can be switched over to encrypted with the STARTTLS command before authentication.
4. For forced use of the STARTTLS command set **prot** to "pop3" and **starttls** to "force". The client will have to send a STARTTLS command before he is allowed to do anything else.

Certificate files must be stored in the directory named by the *`tlsdir`* variable. The files must have the name of the virtual server they are used for plus a ".pem" extension. Each file contains the RSA private key and a certificate for this virtual server.

See the **prng** for instructions how to seed the pseudo random number generator.

SSLv2 is considered insecure and disabled by default. Use the *`allowsslv2`* variable to change this behaviour.

The connection between **pproxy** and **pserv** is always unencrypted. Use a secure tunnel if they are not in the same LAN.

10.2. Usage of the user 'root'

Usage of the user 'root' is kept to a minimum. The only program that needs to run as 'root' is the **ringd** server that is used by **pproxy** to bind sockets to low TCP ports. Everything else runs with the privileges of a normal user.

10.3. Input checks

All input from network sockets is checked for ASCII NUL ('\0') characters and the connection is immediately and silently dropped if a NUL character is found. This event is logged as message 0x0035 ('null_byte_in_input'). A NUL byte can never be part of a valid POP3 dialogue.

pserv and **pcheckd** check the mailbox names coming from **pproxy**. Only the following characters are allowed in mailbox names: a-Z, A-Z, 0-9, '.' (dot), '_' (underscore), '-' (hyphen), '+' (plus), '/' (forward slash), '=' (equal sign), and '%' (percent). Two adjacent dots ("..") are not allowed.

10.4. Using `chroot`

I see no reason why *POPular* can't be run in a `chroot` environment, although I haven't tried it. In the `chroot` environment you need the binaries of the servers and any shared libraries they use, the log directory (`/var/log/popular`), and the run directory (`/var/run/popular`). Depending on your configuration, you might need some config files from `/etc/popular`. For the proxy all files needed for authentication have to be included and for the storage server all mailbox directories. If you have the mailboxes on several disks, you have to mount them all inside the `chroot` environment.

One feature of *POPular* is going to make difficulties: The server reads `/proc/loadavg` to determine the load and react accordingly. It is probably not a good idea to mount `/proc` in the `chroot` environment, so you either have to live without the feature or find some way around the limitation of not being able to read `/proc/loadavg`.

That said, I don't really see much reason for going through all the hassle of setting up and maintaining the `chroot` environment. All the important data, that you want to protect either has to be in the `chroot` environment anyway (like the mailboxes) or at least has to be accessible from it through the network (like authentication data). Of course, it will be harder for an attacker, but it will be a bit harder for the sysadmin, too. Decide for yourself, whether you want that extra bit of security.

If somebody is using *POPular* in a `chroot` environment, I like to hear from you. Especially if I need to put some changes into *POPular* to make it easier to use in a `chroot` environment.

Chapter 11. Standards compliance

POPular implements the 'Post Office Protocol - Version 3' as described by RFC 1939. The optional commands TOP and UIDL from RFC 1939 are supported. The APOP command is not supported.

The *POPular* implementation differs from RFC 1939 in respect to mailbox locking. Please see the chapter on Mailbox locking for details.

POPular is compliant to RFC 2449 ('POP3 Extension Mechanism'). It supports the following capabilities: TOP, UIDL, USER, and PIPELINING. It does not support SASL, RESP-CODES, and LOGIN-DELAY. Please see the chapter on POP3 extensions and capabilities for details.

POPular does not support the LAST command, which was described in older RFCs, but was removed in RFC 1725 in November 1994 and is not even mentioned in the current RFC 1939.

Of the three possible authentication mechanisms (USER/PASS and APOP described in RFC 1939 and the AUTH command described in RFC 1734 and RFC 2195) only USER/PASS is supported.

Strictly speaking a POP3 server is not allowed (RFC 1939) to answer a connection attempt with a negativ response ('-ERR'). *POPular* can do this anyway, if you want to take down the server for maintainance for instance. Alternatively you can configure *POPular* not to accept the connection at all or fake an empty mailbox, which would both be conforming to the RFC.

The POP URL Scheme (RFC 2384) is not needed in POPular.

The following RFCs concerning POP are old and don't apply any more: RFC 918, RFC 937, RFC 1081, RFC 1082, RFC 1225, RFC 1460, RFC 1725, RFC 1957, RFC 2095.

The RFC 2595 ('Using TLS with IMAP, POP3 and ACAP') is supported. See the TLS information in the security chapter for details.

Chapter 12. Internals

12.1. The extended Maildir format

The *POPular* POP server uses a slightly modified Maildir format for storing mailboxes. The format is for most uses compatible to the original mailbox format, allowing standard software components to be used.

In contrast to the original Maildir format the line ending used is CRLF and not only LF. Both SMTP and POP3 use CRLF as a line ending and there is no sense converting to LF endings when saving the mail and back to CRLF when reading the mail. The added advantage is that the size of the file is exactly the file size that is reported in POP3 or with the ESMTP SIZE extension in SMTP.

With the Maildir format each mail in a mailbox is saved in its own file instead of all mails together in one file. A mailbox is a directory with the three subdirectories `new`, `cur` and `tmp`. New, unread mails are in the `new` directory. Read mails are in the `cur` directory. The `tmp` is used for delivery. A mail is saved in this directory with a unique name. Only after the mail has been written completely to disk it is moved to the `new` directory. With this scheme no locking or copying of mails or mailboxes is needed when delivering, reading or deleting mails.

The filenames used for the mails must be unique. Somewhat differently than in the original description of the Maildir format the following naming scheme is used: A filename has three components, the UNIX system time (in seconds after 1970/1/1), the process id of the process delivering the mail and the size of the mail. The components are separated by a dot (.) and an underscore (_), respectively. An example for a file name would be `945113196.14341_1067`. The size of the mail is saved in the filename, so that we don't have to stat the file in order to find out about its size.

With the original Maildir format, the hostname of the delivering system is written into the filename. This is only needed if NFS is used for mailbox access or mail files are copied between hosts to the same mailbox. Because we don't do any of this, we don't put the hostname in there.

A Status-Header is never saved inside a mail message file, because it would mean that the file has to be rewritten after it is RETRieved, but not DELEted. Instead, **pserv** will add a Status-Header on the fly while sending the mail out if the `statusheader` configuration variable is set and the mail file is in the `cur` directory.

POPular can cope with several pathologic cases of mail file contents like totally empty mails, mails without headers etc. Empty mails will not be accessible, an error is logged. Other strange mails are accessible, but the client might not understand it. Mails that don't end in CRLF have a CRLF appended, so that the end-of-file-marker 'CRLF.CRLF' is recognized. The appended CRLF don't count towards the mail size.

12.2. The XPOP protocol

The *POPular* POP proxy and POP servers communicate through the XPOP protocol, which is a slightly modified POP3 protocol (defined in RFC 1939 [RFC1939]). The modifications are needed for telling the POP server the mailbox to use and some more information. This extension is used only by the *POPular* proxy and server. If you use a different server, use the POP3 protocol.

The XPOP protocol is really simple. As soon as the connection between the proxy and the server is established the proxy sends three lines of text, each ending with a CRLF combination. The content of the lines is, in this order:

1. The mailbox to use. This is a directory name relative to the POP spool defined in the configuration of the POP server. It can include any number of subdirectories. It does not start with a `/`. No ".." is allowed in the name.

2. An ID string used for this connection. This ID is generated by the POP proxy and used in all logging messages on the proxy and server to be able to easily find corresponding entries.
3. A list of flags. Flags are denoted by ASCII chars. In the absence of the char, the flag is ’false’, otherwise ’true’. Currently there is only one flag ’M’ defined.

After this three lines are sent the POP server answers with ’+OK’ or ’-ERR’ according to the POP3 protocol. This message is passed through to the client. From then on everything is just like in the POP3 protocol but starting in the transaction phase, not in the authorization phase.

When configuring a backend server the protocol CXPOP can be used. This just means that the XPOP protocol should be used after checking for available mail with the MAILCHECK protocol, which is described in the next chapter.

12.3. The MAILCHECK protocol

On every storage server a **pcheckd** program is running. It implements the server side of the MAILCHECK protocol. The client side is implemented in the **pproxy** program. There is a test program called **pcheck**, which fully implements the protocol. It can be called from the command line.

The MAILCHECK protocol uses a single UDP request packet and a single UDP packet with the answer. The request packet contains with an asterisk (’*’) a single uppercase letter denoting the type of request, extra parameters and an optional carriage return (CR) and an optional line feed (LF) character.

Currently there are two types of requests defined:

Load check ’L’

The request contains only the letter ’L’. It asks the server to send back the current load of the machine. The value is returned as a floating point value formattet in ASCII. It is intended to be used by the proxy server to detect storage server overload. (Although, in the current version of the proxy, it is not used.)

Example: reqest: ’*L’ answer: ’+OK 3.23\n’

Mailbox check ’M’

The request contains the letter ’M’ followed directly by the mailbox name including any directory names relative to the POP storage directory. It asks the server to return the current state of the mailbox, i.e. whether there are any mails in the mailbox.

Example: request: ’*M3/2d/joe.doe’ answer: ’+OK 0 no mails\n’

Note: For compatibility with earlier versions of this programm a slightly different syntax is allowed. If the first character is not an asterisk (’*’), the request is assumed to be a mailbox check and the only content of the packet is the mailbox. Support for this format will be discontinued in a future version.

The answer is a single UDP packet formatted as follows: If an error has occured on the server side it will return ’-ERR ’, an error message and a line feed character. If there was no error the server will return ’+OK ’ and an additional message as follows:

'+OK 1'

This is the answer from the Load command.

'+OK 0 no mail'

There are no mails in this mailbox.

'+OK 1 mail'

There is no new mail, but only read mail in this mailbox.

'+OK 2 new mail'

The is at least one new mail in this mailbox.

'+OK 3 load too high'

The load on the storage server is too high. The proxy will fake an empty mailbox to the client, so the storage server will not be overloaded.

'+OK 4 maxsession reached'

The maximum number of **pserv** sessions is reached. The proxy will fake an emtpy mailbox to the client, so the storage server will not be overloaded.

12.4. Mailbox locking

RFC 1939 describing the POP protocol askes for the mailbox to be locked after authentication of the user and before any access to the mailbox is granted. The mailbox stays locked until after the any changes are done to the mailbox after the client sends the QUIT command.

If mailboxes are saved in any of the formats where all the mails are saved in one big file, locking this file is actually necessary for insuring mailbox integrity. However, if mails are saved in one file per message (maildir format as *POPular* uses it), this is not strictly necessary.

If no locking is used the worst thing that can happen is that a user connects to a mailbox twice (or more times) and sees a message in the message list and when he tries to retrieve the message he will get an error message. While this is obviously not the best solution it isn't that bad.

On the other hand there is no reason why anyone will open several POP connections to the same mailbox. The only common case where this happens is when a POP connection breaks down on the client side, but the server hasn't noticed that yet. This can happen either because the client program or system crashes or the network connection (often going through a modem) fails.

In this situation not locking the mailbox will actually help. The first connection is inactive anyways and having a lock means that the server has to wait for a timeout before it unlocks the mailbox. Only after a timeout the mailbox is accessable again. Without locking the mailbox will be accessible immediately with no ill effects.

To make matters worse, indicating a 'mailbox locked' error to the user will often confuse the user and will cause him to call the providers helpline reporting a problem that has probably solved itself by the time the helpdesk knows about it.

For this reason there is no mailbox locking in **pserv**.

12.5. TCP keepalive option

On all TCP sockets used by **pproxy** and **pserv** the `SO_KEEPALIVE` socket option is set. The default timeout is 2 hours, it can be set in most UNIX systems through a kernel parameter and will then affect all TCP connections with the option set.

On Linux systems the timeout can be set by writing the time in seconds to the file `/proc/sys/net/ipv4/tcp_keepalive_time`.

POP connections are shortlived, if there is no data flowing, it doesn't make sense to keep the connection open. A short keepalive timeout will not produce much more traffic. On the other hand POP connections at large ISPs are often hanging, because clients behind dialups sometimes shut down modem connection without properly closing all TCP connections (or the modem connection might have failed). A short keepalive timeout will help detect these cases and free resources on the proxy and server.

If there are no other services running on the same machines as the POPular services, that might be negatively affected, it is recommended to set the keepalive timeout to a value somewhere between 10 and 30 minutes.

Note that connections where data is still flowing (even if flowing slowly) are not affected by this timer, as the timer is reset on each packet that is sent or received on this connection.

12.6. POP3 extensions and capabilities

RFC 2449 describes a POP3 extension mechanism. A client can use the CAPA command to ask the server about any optional or enhanced capabilities. In the context of *POPular* there is one special problem with capabilities: The CAPA command can be send before or after authentication. But that means that the proxy and the server both have to be able to send the right answer back. To not confuse the client the answer in both cases must be the same or similar (for details see RFC 2449). Because the *POPular* proxy can be used with any server and it doesn't know about its capabilities it can't send a proper answer to a CAPA request on its own.

To solve this problem the proxy and the server can be configured to answer anything you like to a CAPA query. For different virtual servers, different CAPA listings can be saved. Note that, because a CAPA request can be sent before authentification, it is not possible to bind a list of capabilities to a backend server.

Setting up a capabilities list happens in two steps. First you edit a file in `/etc/popular/capa` (the directory can be changed through **pcontrol** with **set capadir**) and put all the capabilities you want to offer in that file. You can give the file any name you want. This name will later be used as the ID of this capability list in **pcontrol**.

The format of the capabilities file is easy. Just list the capabilities one per line, exactly as they are supposed to appear as answer to the CAPA command. Have a look at RFC 2449 for the details. If you get this wrong, the answer to the CAPA command might be not conforming to RFC 2449! Use a normal LF as line ending, *POPular* will automatically translate that to a proper CRLF line ending for the POP protocol. Note that there is a compiled in upper limit (MAXLEN_CAPA) on the length of all the capabilities joined together in one list.

After you have set up the file (or files) you can load the capabilities list into **pproxy** with the **pcontrol** utility. With **capa load FILENAME** the file is loaded. **capa show NAME** will show you the loaded capability list. Note that for printing purposes the list of capabilities will appear on one line, separated by commas. Capability lists can be reloaded at any time with the **capa load** command and deleted with **capa del NAME**. With **capa list** you can get a list of all currently loaded capability lists.

There are three special capability lists build into *POPular*: ERROR, NONE and DEFAULT. ERROR means to answer with an error to any CAPA command sent by the client, emulating an old server without CAPA support. NONE will send an empty capabilities list and DEFAULT will send a small list of capabilities that are supported by both **pproxy** and **pserv**. That means that if you use only *POPular* programs and no external servers you can use DEFAULT.

The last step is assigning a capability list to each virtual server. This can be done with the **vserv conf VSERV capa CAPABILITY-ID** command. You can set any of the three builtin capability lists or any of the ones you loaded.

The following capabilities are always supported by *POPular*: TOP, UIDL, USER and PIPELINING. SASL, RESP-CODES, and LOGIN-DELAY are not supported. The EXPIRE capability is outside the scope of the *POPular* software, but you might want to use it. The default capability list does not include the IMPLEMENTATION capability. If you want to use it, please use 'IMPLEMENTATION POPular-<version>'.

Chapter 13. *POPular* system design issues

13.1. High availability issues

If you want to build a real high available mail system with the *POPular* server suite, some things have to be considered. The *POPular* proxy and server in itself are not enough to get a highly available system. This documenation can't go into all details of building a highly available system, but some issues to get you startet will be discussed.

The POP proxy

For high availability you want to install more than one server running the *POPular* proxy software. You will need some way of failing over the IP address to the other server. Depending on your configuration of the proxy server you will have to change the configuration when this failover occures.

The storage server

Each storage server has access to only a subset of all mailboxes. If a storage server fails, no access to this subset is possible anymore. All the mailboxes stored on other servers are unaffected. There is no good way around this. Even if you use a big dedicated NFS server for storing all the mailboxes, this server can fail. The only way to make this really redundant would be by saving all the mails on more than one hard drive attached to separate hosts, which would induce a hugh overhead of transaction processing to keep the two copies in sync. For most uses the best solution is probably to use a multihost capable RAID system for storing the mails. This RAID system is attached to more than one host. In case of failure of one host the other host attached to the same RAID mounts the disks of the failed host after doing a file system check. A similar solution could be build with a SAN (storage area network) based on fibrechannel or a similar technology.

Whichever system you use, when failing over to another storage server, the proxy servers needs to know which server to access. You can either move the IP number of the failed storage server over to the new server or change the configuration of the proxy server to the new IP number or host name.

Independently of the exact storage architecture used, you obviously want to use a RAID system instead of simple disks. Disks are by far the most unreliable part of any system and you should be sure to handle disk failures without downtimes.

Mail delivery

Exactly how you get the mail delivered to the storage server is not part of this documentation. Depending on what mechanism is used for deciding which mail to deliver where, the fail over of a storage server has to be taken into account.

13.2. Mailbox directory layout

The *POPular* software doesn't force you to use any particular layout for the mailbox directories. The simplest layout would use a `/var/spool/mail` (or similar) directory with subdirectories for all the mailboxes. This layout has the severe drawback that it doesn't scale well. A directory with thousands of files is slow to access, especially on some filesystems like the Linux Ext2fs. Instead a hierarchical structure is recommended. Depending on the anticipated maximal number of mailboxes in the system the hierarchy has to have more or less levels.

There are several possibilities how to distribute the mailboxes into subdirectories. The obvious way is to use the mailbox name (or part of it) as a name for the subdirectory. This works well if the mailbox names are sufficiently "random", so that the mailboxes are distributed evenly. In most real life situations this is not the case. And even if the mailboxes in your case have a known structure and nice random names this might change over time.

An easy technique is to calculate a hash of the mailbox name and use this to name the subdirectories. The MD5 hash algorithm has proven to be very good for this. Whatever mailbox names you throw at it, the hash is very evenly distributed. And there are lots of implementations for every language available. Using one or two hex numbers from the MD5 hash for every directory level will result in directories with not more than 16 or 256 entries, respectively.

13.3. Performance tuning

It is probably a good idea to use the 'noatime' mount option on the mailbox storage disks.

13.4. Using virtual servers

Appendix A. Man pages of *POPular* commands

The information in this chapter is also contained in the individual manpages of every command.

pcheck

Name

`pcheck` — popcheck client

Synopsis

pcheck [`-s` *`SERVER`*`, --server=`*`SERVER`*] [`-p` *`PORT`*`, --port=`*`PORT`*] [`-t` *`TIMEOUT`*`,`
`--timeout=`*`TIMEOUT`*] [`--help`] [`--version`] [`-l, --get-load` | `-m` *`MAILBOX`*`,`
`--get-mailbox=`*`MAILBOX`*]

Description

This command is used for checking the proper function of the **pcheckd**. It will send a UDP request to the **pcheckd** running on the machine *`server`* (default: 127.0.0.1) on port *`port`* (default: 50110). It will wait for *`timeout`* seconds for an answer. The answer is printed on stdout.

There are two types of requests: The mailbox request (-m) will ask the server about the state of the named mailbox and will output it ('0 no mail', '1 mail', '2 new mail', or '3 load too high'). The load request (-l) will ask the server for the load avarage for the last minute.

The output of the command is the answer to the request returned by the server. It has the general format '+OK some message' if the request was successfull or '-ERR error message' if some error occured on the server side.

Options

--help

Get help on parameters.

--version

Print version information.

-s *`server`*

Host name or IP address of POPCHECK server.

-p *`port`*

UDP port for POPCHECK service on server.

-t *timeout*

Timeout in seconds when waiting for answer.

-l

Return load avarage of server.

-m *mailbox*

Name of mailbox to check.

Return Codes

0

Successful, 'no mails' when using option -m.

1

Successful, 'mails available' when using option -m.

2

Error connecting to server, other system errors or "-ERR" msg from server

3

Error while parsing the command line.

See Also

pcheckd(8), pclean(8), pcontrol(1), pdeliver(1), pproxy(8), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

pcheckd

Name

`pcheckd` — Checks for mail in maildir mailboxes

Synopsis

pcheckd [`-l` *FILE*, `--logfile` *FILE*][`-m` *DIR*, `--mailboxdir` *DIR*][`-p` *PORT*, `--port` *PORT*][`-e` *LOAD*, `--emptyload` *LOAD*][`-f` *NUM*, `--fork` *NUM*][`-r` *DIR*, `--rundir` *DIR*][`-h` *HOST*, `--host` *HOST*][`--nodeamon`][`--debug`][`--help`][`--version`]

Description

This program will open the named UDP port (default: 50110) and listen for requests. It will send an answer to the requests back to the client.

There are two types of requests: The mailbox request ('M') will ask the server about the state of the named mailbox ('0 no mail', '1 mail', '2 new mail', or '3 load too high'). The load request ('L') will ask the server for the load avarage for the last minute.

The load is returned as an integer which is determined by reading /proc/loadavg and cutting of everything after the decimal point.

This server is single threaded.

Options

--help

Print short command line help.

--version

Print version information.

-l *FILE*, --logfile *FILE*

Name of logfile (default: `/var/log/popular/pcheckd`)

-m *DIR* --mailboxdir *DIR*

Directory where all the mailboxes reside. There can be an arbitrary depth of subdirectories beneath this. All mailbox names sent by the client are relative to this directory.

-p *PORT*, --port *PORT*

UDP port to listen to.

-e *LOAD*, --emptyload *LOAD*

If the system load average over the last minute is LOAD or higher, a 'M' request will be answered with '3 load too high'. In this case **pproxy** will fake an empty mailbox.

-f *NUM*, --fork *NUM*

Fork NUM processes at start of pcheckd. The parent and all the children will answer requests. The parent doesn't count towards NUM, so there will be NUM+1 pcheckd processes.

-r *DIR*, --rundir *DIR*

Name of run directory. The pid file is saved into this directory. (default: `/var/run/popular`)

-h *HOST*, --host *HOST*

Host name or IP number to bind to. This might be needed if you have several IP numbers on one host. Default is to bind to INADDR_ANY (0.0.0.0), i.e. to listen to all interfaces.

--nodeamon

Don't run as daemon in the background.

--debug

Enable debug logging. This will log all packets as they are received and sent.

See Also

pcheck(1), pclean(8), pcontrol(1), pdeliver(1), pproxy(8), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

pclean

Name

`pclean` — Script for periodic cleanup jobs

Synopsis

pclean `[-h]` `[-v]` `[-V]` `[-D]` `[-t]` `[-e]` `[-s]` `[-c]` `[-m]` `[-b]` `[-n]` `[-f` *`file`*`]`

Description

This script takes care of several periodic cleanup jobs. It should be run nightly.

Options

-h

Get help on parameters.

-v

Print version information.

-D

Print debug messages (Be careful! This is really verbose)

-t

Check tmp directories for old files

-e

Expire mailbox contents

-s

Check and report size of all mailboxes

-c

Check names, ownership, and permissions of all dirs/files

-m

Put mail in mailboxes that are over quota (implies -c)

-b

Backup deleted mailboxes

-f file

Read options from this file

-n

Don't do anything, just print what you would do (implies -D)

See Also

pcheck(1), pclean(8), pcontrol(1), pdeliver(1), pproxy(8), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

pcontrol

Name

`pcontrol` — Send commands to running pproxy or pserv

Synopsis

pcontrol [`-c` *CMD*`, --command=`*CMD*] [`-s` *FILE*`, --socket=`*FILE*] [`-p` *PRG*`, --program=`*PRG*] [`-q, --quiet`] [`-v, --verbose`] [`-i, --ignore-errors`] [`--help`] [`--version`] [*FILE*`...`]

Description

With **pcontrol** commands can be send to the running **pproxy** or **pserv** server. If no files are given on the command line and the `--command` option is not used, commands are read interactively from the terminal. If GNU readline support is compiled in, it will be used.

If one or more filenames are given on the command line, these files are read in order and all lines are send to the server. Lines beginning with '#' are ignored. '-' can be used to read from STDIN.

If the `--command` option is used, the command after `--command` is send to the server.

If neither the `--program` nor the `--socket` option is given, the program will look for the socket of a running **pproxy** and if that is not found for the socket of a running **pserv**.

Start the program in interactive mode and type '/h' for more information about possible commands, parameters and return codes.

Options

-c *CMD*, --command=*CMD*

Send CMD to server instead of reading them from the named files.

-s *FILE*, --socket=*FILE*

Open FILE as control socket to server. This option can not be used together with `--program`.

-p *PRG*, --program=*PRG*

Open control socket to program PRG. The corresponding socket in `/var/run/popular` will be used. This option can not be used together witch `--socket`.

-q, --quiet

Don't print answer messages from server.

-v, --verbose

Print messages as they are sent to the server.

-i, --ignore-errors

Ignore errors from server. Without this option the program is terminated after the first error and a corresponding return code is set.

--help

Print short help screen.

--version

Print version information.

See Also

pcheck(1), pcheckd(8), pclean(8), pdeliver(1), pproxy(8), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

pdeliver

Name

`pdeliver` — Deliver mail into a maildir mailbox

Synopsis

pdeliver [--help][--version][-q *QUOTA*, --quota *QUOTA*][-Q *NUM*, --maxnum *NUM*][-m *DIR*, --mailboxdir *DIR*][-f *FORMAT*, --format *FORMAT*][-l *FILE*, --logfile *FILE*][-i *ID*, --id *ID*][*mailbox*]

Description

This command awaits a mail on STDIN formatted according to the format specified with the `-f` option. It will write the mail to the mailbox specified on the command line in maildir format.

Options

--help

Get help on parameters.

--version

Print version information.

-q *QUOTA*, --quota *QUOTA*

This is the quota for the total size of the mailbox in MBytes (1024*1024 bytes) or 0, if the total size of the mailbox should not be checked. Default is 0.

-Q *NUM*, --maxnum *NUM*

This is the quota for the number of mails in the mailbox or 0, if the check should not be performed. Default is 0.

-m *DIR*, --mailboxdir *DIR*

This is the directory where all mailbox directories are. Default is `/pop`.

-f *FORMAT*, --format *FORMAT*

Format of the mail an STDIN. There are three mail formats understood by **pdeliver**:

LF

Each line in the mail ends in a line feed (LF) character. This is the common format for UNIX text files.

CRLF

Each line in the mail ends in a carriage return (CR), line feed (LF) combination.

NET

Each line in the mail ends in a carriage return (CR, line feed (LF) combination. Lines beginning with a dot (.), have this dot doubled. A line consisting of a single dot and a carriage return (CR), line feed (LF) combination denotes the end of the mail. **pdeliver** will stop reading at this point.

Irrespective of the format an EOF will always end the mail.

-l *FILE*, --logfile *FILE*

Logfile used for all messages. Default is `/var/log/popular/pdeliver`. if you don't want any logging, you have to use '-l `/dev/null`'.

-i *ID*, --id *ID*

This ID is logged with every line in the log file to find corresponding entries. If you don't specify an ID "-" will be logged. You probabely want to put the internal ID of the MTA here to easily find corresponding entries in the MTA log file.

Quota

The quota will be checked before delivering the mail. The size of the mail currently being delivered is not counted towards the mailbox size. If the mailbox size is smaller then the quota and the number of mails smaller then the maximal allowed number of mails, the mail is delivered, otherwise **pdeliver** exits with return code 1.

Because there is no locking done, two processes delivering into the same mailbox at the same time might lead to overfull mailboxes.

Return codes

0

Mail was successfully delivered.

1

Quota for this mailbox exceeded. Mail is not delivered.

2

Error while delivering. Permission problems or something like that.

3

Command line error.

See Also

pcheck(1), pcheckd(8), pclean(8), pcontrol(1), pproxy(8), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

pproxy

Name

`pproxy` — A POP3 proxy server

Synopsis

pproxy `[--help]` `[--version]` `[ -l FILE, --logfile FILE ]` `[ -r DIR, --rundir DIR ]`

Description

pproxy is a POP3 proxy. It will wait for username and password from the POP3 client and authenticate the user. It will then ask the pcheckd(8) deamon on the POP3 server whether the mailbox is empty. If it is empty it will keep talking to the client, "simulating" the empty mailbox. If there is mail in the mailbox it will connect to the pserv(8) POP-Server, tell it which mailbox to open and then just forward data back and forth between the client and the server.

For details see the POPular manual.

Options

--help

Print short help screen.

--version

Print version information.

-l *FILE*, --logfile *FILE*

Logfile to use. Default is `/var/log/popular/pproxy`.

-r *DIR*, --rundir *DIR*

Name of run directory. The pid and state file, and the control socket are saved into this directory. (default: `/var/run/popular`)

Files

`/var/log/popular/pproxy`

Log file.

See Also

pcheck(1), pcheckd(8), pclean(8), pcontrol(1), pdeliver(1), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

pserv

Name

`pserv` — A POP3 server

Synopsis

pserv [`--help`] [`--version`] [`-l` *`FILE`*, `--logfile` *`FILE`*] [`-r` *`DIR`*, `--rundir` *`DIR`*]

Description

This is a modified POP3 server without the authentication part. It is supposed to be used in conjunction with the POPular **pproxy** server.

The **pserv** program is intended to be run as normal user, who should have all necessary permissions to access the mailbox files.

Options

--help

Print short help screen.

--version

Print version information.

-l *`FILE`*, --logfile *`FILE`*

Logfile to use. Default is `/var/log/popular/pserv`.

-r *`DIR`*, --rundir *`DIR`*

Name of run directory. The pid and state file, and the control socket are saved into this directory. (default: `/var/run/popular`)

See Also

pcheck(1), pcheckd(8), pclean(8), pcontrol(1), pdeliver(1), pproxy(8), pstatus(1), ptestpdm(1), The POPular Manual

pstatus

Name

`pstatus` — Display status of POPular proxy and server

Synopsis

pstatus [`-a, --active`][`--help`][`-f` *`FILE`*`, --file` *`FILE`*][`-l` *`TIME`*`, --loop` *`TIME`*][`-p` *`PRG`*`, --program` *`PRG`*][`-r` *`RANGE`*`, --range` *`RANGE`*][`--version`]

Description

This command dumps the status information of a running **pproxy** or **pserv**, respectively. The data is dumped on STDOUT in a human readable form. In both cases the list of sessions with current data and statistical data is dumped.

Options

-a, --active

Print only active session slots.

-f *FILE*, --file *FILE*

Filename to read the data from. Normally this is `/var/run/popular/pproxy.state` or `/var/run/popular/pserv.state`. If no file name is given and the `--program` option is not used, both these names are tried and the first one found is used. This option can't be used together with the `--program` option.

--help

Print short help screen.

-l *TIME*, --loop *TIME*

Print data in a loop every TIME seconds. Default is no looping. If no TIME is given, one second is used.

-p *PRG*, --program *PRG*

Program to read data from. Can be **pproxy** or **pserv**. The corresponding `.state` in `/var/run/popular` is read. This option can't be used together with the `--file` option.

-r *RANGE*, --range *RANGE*

Print only the range of slots. RANGE can be one of 'n-m', 'n-', or '-m'.

--version

Print version information.

See Also

pcheck(1), pcheckd(8), pclean(8), pcontrol(1), pdeliver(1), pproxy(8), pserv(8), ptestpdm(1), The POPular Manual

ptestpdm

Name

`ptestpdm` — Test POPular Database Modules (PDMs)

Synopsis

ptestpdm [`--help`] [`--version`] [`--debug`] [`-d` *`DIR`*, `--dir` *`DIR`*] [*`COMMAND-FILE`*] [*`DATA-FILE`*]

Description

The **ptestpdm** is used for testing the POPular Database Modules. The COMMAND-FILE contains a list of module definitions in the format ID MODULE-NAME [ARGS] ... The DATA-FILE contains a list of users to check in the format IP PROTOCOL NAMESPACE USER PASSWORD. Empty lines and comment lines (starting with '#') are allowed.

ptestpdm will read the COMMAND-FILE and load and initialize all modules. Then it reads the DATA-FILE and checks all users against the loaded modules. The result of this check is printed on stdout.

Options

--help

Print short help screen.

--version

Print version information.

--debug

Enable debug messages.

-d *`DIR`*, --dir *`DIR`*

Directory from which the modules are to be read. Default is `/usr/local/lib/popular`.

See Also

pcheck(1), pcheckd(8), pclean(8), pcontrol(1), pdeliver(1), pproxy(8), pserv(8), pstatus(1), The POPular Manual

ringd

Name

`ringd` — Server for binding low TCP ports from non-root programs

Synopsis

ringd `[--help][--version][ -u USER, --user USER ][ -g GROUP, --group GROUP ]`

Description

ringd is used by **pproxy** to open TCP ports <1024 for listening. **ringd** runs in the background and gets requests to open a port through a UNIX domain socket. It will then open a socket, bind it to the specified port and pass it back to the calling program. **ringd** also stores the file descriptor itself and if another process askes for the same port it will just hand over the already bound socket. This way multiple independent processes can share the same bound port. **ringd** keeps a refcount so it knows when to close the socket itself, but this means that all processes, that got a socket through **ringd**, must make sure to tell **ringd** when they close the socket or exit.

ringd will reopen its log file if it receives a HUP signal.

For details see the POPular manual.

Options

--help

Print short help screen.

--version

Print version information.

-u *USER*, --user *USER*

User which is used for ownership of the Unix domain socket which is used to talk to the pproxy. This should be the same user pproxy is running as. No default.

-g *GROUP*, --group *GROUP*

Group which is used for ownership of the Unix domain socket which is used to talk to the pproxy. This should be the same group pproxy is running as. Default is the primary group of the given user.

Files

`/var/log/popular/ringd`

Log file.

See Also

pcheck(1), pcheckd(8), pclean(8), pcontrol(1), pdeliver(1), pserv(8), pstatus(1), ptestpdm(1), The POPular Manual

Appendix B. The PDM C API

Note: I am not perfectly happy with this interface, so there might be some changes in the future.

PDM modules are shared libraries. Each module must supply the following functions: `pdm_init`, `pdm_close`, `pdm_args`, and `pdm_reload`. In addition it must supply at least one of the functions `pdm_check` or `pdm_auth`.

The functions must be defined as follows:

```
char *pdm_init(int argc, char *argv[], struct pdm_mvar *mvar, void **pdmctx);
```

The `pdm_init` is called once when the module is loaded. All arguments are supplied in *argv*, the number of arguments is in *argc*. The first argument is always the ID (symbolical name) of this module, the second argument is the name of the module (without the prefix "libpdm_" and suffix ".so"). The meaning of the other arguments can be defined by each module according to its needs, for instance it could be used for filenames, the host name of a database and so on.

The parameter *pdmctx* is a pointer to a void pointer. The module can store a pointer to arbitrary data in this void pointer. This data will not be used by *POPular* itself, but will be returned to the module with all other function calls.

The function must return NULL if the module was successfully initialized or an error string otherwise.

```
int pdm_close(void *pdmctx);
```

If the module isn't needed anymore, *POPular* will call the `close` function. It's only parameter is the PDM context set by the `init` function. The module should close all files, close networks sockets, deallocate memory, and generally do all necessary cleanup. After this function returns *POPular* will unload the shared library.

```
char *pdm_args(void *pdmctx);
```

If *POPular* needs to know with what arguments a module was initialized, it calls this function. The function must return a string which is the same or equivalent to the string with which the module was originally configured. Normally this is the concatenation of the arguments separated by spaces.

```
int pdm_reload(void *pdmctx);
```

This call instructs the module to reinitialize itself. If it has any files or network connections open, it should close and reopen them, etc.

```
pdm_result_t pdm_check(void *pdmctx, const struct pdm_request *pdmr, struct
pdm_data *pdmd);
```

This function is used to ask the database for a specific user. The PDM context is supplied as well as all available information about the user in the *pdmr*. The `pdm_check` must return the result of the lookup and return additional information through the *pdmd*. See below for a description of the pdm_request and pdm_data structures.

```
pdm_result_t pdm_auth(void *pdmctx, const struct pdm_request *pdmr, struct pdm_data
*pdmd);
```

This function is used to check a password through the database module. If the user and all the data associated with the user was already established by another module through the `pdm_check` call, but the user was not authenticated, then this module will be called.

For communication between *POPular* and the PDM modules two C structs are used:

```
struct pdm_request {
  char *peer;
  protocol_t prot;
  char *user;
  char *pass;
  char *namespace;
};
```

This struct is used to tell the `pdm_check` and `pdm_auth` functions about a user who is to be authenticated. It contains the IP address of the client in dotted quad notation in *peer*, the protocol the client is using in *prot* (which can be used to find out whether the session is encrypted), and the username, password and namespace in *user*, *pass*, and *namespace*, respectively. Usually only the last three fields will be used for authenticating a user, but if the module wishes to do so, it can use all fields.

```
struct pdm_data {
  int flags;
  pdm_fail_reason_t reason;
  char backend[MAXLEN_ID];
  char user[MAXLEN_USERNAME];
  char pass[MAXLEN_PASSWORD];
};
```

The pdm_data struct is used by the `pdm_check` to return details about this user that have been found in the database. The *backend* field contains the ID of the backend server this users mailbox is on. If this is a normal POP3 server the fields *user* and *pass* contain the username and password, respectively, that are to be used to authenticate to that server. If the *POPular* server is used, the *user* field contains the directory name of the mailbox. If the lookup failed, detailed information about the reason should be put into *reason*. This information will be logged, but is not available to the user. The *flags*, finally, can be used to set special flags. Currently this is only used for marking a user as authenticated with a master password instead of his normal password. See the description of the *POPular* server for the meaning of this flag.

Appendix C. Copyright