Crossroads 1.38

Karel Kubat
e-tunity

2005, 2006, ff.

Abstract

Crossroads is a load balance and fail over utility for TCP based services. It is a daemon program running in user space, and features extensive configurability, polling of back ends using 'wakeup calls', detailed status reporting, 'hooks' for special actions when backend calls fail, and much more. Crossroads is service-independent: it is usable for HTTP/HTTPS, SSH, SMTP, DNS, etc. In the case of HTTP balancing, Crossroads can modify HTTP headers, e.g. to provide 'session stickiness' for back-end processes that need sessions, but aren't session-aware of other back-ends.

Contents

1 Introduction

Crossroads is a daemon that basically accepts TCP connections at preconfigured ports, and given a list of ’back ends’ distributes each incoming connection to one of the back ends, so that a client request is served. Additionally, crossroads maintains an internal administration of the back end connectivity: if a back end isn’t usable, then the client request is handled using another back end. Crossroads will then periodically check whether a previously not usable back end has come to life yet. Also, crossroads can select back ends by estimating the load, so that balancing is achieved.

Using this approach, crossroads serves as load balancer and fail over utility. Crossroads will very likely not be as reliable as hardware based balancers, since it always will require a server to run on. This server, in turn, may become a new Single Point of Failure (SPOS). However, in situations where cost efficiency is an issue, crossroads may be a good choice. Furthermore, crossroads can be deployed in situations where a hardware based balancing already exists and augmenting service reliability is needed. Or, crossroads may be run off a diskless system, which again improves reliability of the underlying hardware.

This document describes how to use crossroads, how to configure it in order to increase the reliability of your systems, and how to compile the program from its sources.

1.1 Obtaining Crossroads

As quick reference, here are some important URL’s for Crossroads:

- `http:/crossroads.e-tunity.com` is the site that serves Crossroads. You can browse this at leisure for documentation, sources, and so on.
- `http://freshmeat.net/projects/crossr` is the Freshmeat announcement page.
- `svn://svn.e-tunity.com/crossroads` is the SVN repository; anonymous reading (fetching) is allowed. In order to commit changes, `mail me`[1] for credentials.

1.2 Copyright and Disclaimer

Crossroads is distributed as-is, without assumptions of fitness or usability. You are free to use crossroads to your liking. It’s free, and as with everything that’s free: there’s also no warranty.

You are allowed to make modifications to the source code of crossroads, and you are allowed to (re)distribute crossroads, as long as you include this text, all sources, and if applicable: all your modifications, with each distribution.

While you are allowed to make any and all changes to the sources, I would appreciate hearing about them. If the changes concern new functionality or bugfixes, then I’ll include them in a next release, stating full credits. If you want to seriously contribute (to which you are heartily encouraged), then mail me and I’ll get you access to the Crossroads SVN repository, so that you can update and commit as you like.

1.3 Terminology

Throughout this document, the following terms are used: [2]

A client is a process that initiates a network connection to get contact with some service.

A service or **server process** or **listener** is a central application that accepts network connections from clients and sevices them.

---

[1] mailto:karel@e-tunity.com

[2] Many more meanings of the terms will exist – yes, I am aware of that. I’m using the terms here in a very strict sense.

Back ends are locations where crossroads looks in order to service its clients. Crossroads sits 'in between' and does its tricks. Therefore, as far as the back ends are concerned, crossroads behaves like a client. As far as the true client is concerned, crossroads behaves like the service. The communication is however transparent: neither client nor back end are aware of the middle position of crossroads.

A connection is a network conversation between client and service, where data are transferred to and fro. As far as crossroads is concerned, success means that a connection can be established without errors on the network level. Crossroads isn't aware of service pecularities. E.g., when a webserver answers `HTTP/1.0 500 Server Error` then crossroads will see this as a succesful connection, though the user behind a browser may think otherwise.

Back end selection algorithms are methods by which crossroads determines which back end it will talk to next. Crossroads has a number of built-in algorithms, which may be configured per service.

Back end states are the statusses of each back end that is known to crossroads. A back end may be available, (temporarily) unavailble or truly down. When a back end is temporarily unavailable, then crossroads will periodically check whether the back end has come to life yet (that is, if configured so).

A spike is a sudden increase in activity, leading to extra load on a given service. When crossroads is in effect and when the spike occurs in one connection, then obviously the spike will also appear at one of the back ends. However, crossroads will see the spike and will make sure that a subsequent request goes to an other back end. In contrast, when several connections arrive simultaneously and cause a spike, then crossroads will be able to distribute the connections over several back ends, thereby 'flattening out' the increase.

Load balancing means that incoming client requests are distributed over more than just one back end (which wouldn't be the case if you wouldn't be running crossroads). Enabling load balancing is nothing more than duplicating services over more than one back end, and having something (in this case: crossroads) distribute the requests, so that per back end the load doesn't get too high.

An HTTP session is a series of separate network connections that originate from one browser. E.g., to fill the display with text and images, the browser hits a website several times. An HTTP session may even span several screens. E.g., a website registration dialog may involve 3 screens that when called from the same browser, form a logical group of some sort.

Headers or **header lines** are specific parts of an HTTP message. Crossroads has directives to add or modify headers that are part of the request that a browser sends to server, or those that are part of the server.

Session stickiness means that when a browser starts an HTTP dialog, the balancer makes sure that it 'sticks' to the same back end (i.e., subsequent requests from the browser are forced to go to the same back end, instead of being balanced to other ones).

Back end usage is measured by crossroads in order to be able to determine back end selection. Crossroads stores information about the number of active connections, the transferred bytes and about the connection duration. These numbers can be used to estimate which back end is the least used – and therefore, presumably, the best candidate for a new request.

Fail over is almost always used when load balancing is in effect. The distributor of client requests (crossroads of course) can also monitor back ends, so that incase a back end is 'down', it is no longer accessed.

Service downtime normally occurs when a service is switched off. Downtime is obviously avoided when fail over is in effect: a back end can be taken out of service in a controlled manner, without any client noticing it.

1.4 Porting issues for pre-1.21 installations

As of version 1.21, the event-hook directives `onsuccess` and `onfailure` no longer exists.

- Please replace `onsuccess` by `onstart`;
- Please replace `onfailure` bu `onfail`;
- Note that there is a new hook `onend`.

The commands that are run via `onstart`, `onend` or `onfail` are subject to format expansion; e.g., `%1w` is expanded to the weight of the first back end, etc.. See section 4 for details.

1.5 Porting issues for pre-0.26 installations

As of version 0.26 the syntax of the configuration file has changed. In particular:

- The keyword `maxconnections` is now used instead of `maxclients`;
- The keyword `connectiontimeout` is now used instead of `sessiontimeout`.

Therefore when converting configuration files to the new syntax, the above keywords must be changed. (The reason for these changes is that 0.26 introduces *sticky HTTP sessions* that span multiple TCP connections, and the term *session* is used strictly in that sense – and no longer for a TCP connection.)

1.6 Porting issues for pre-1.08 installations

As of version 1.08, the following directives no longer are supported:

- `insertstickycookie` was replaced by the more generic directive `addclientheader`. E.g., instead of
  `insertstickycookie "XRID=100; Path=/";`
  the syntax is now
  `addclientheader "Set-Cookie:  XRID=100; Path=/";`
- `insertrealip` was replaced by the more generic directive `setserverheader`. E.g., instead of
  `insertrealip on;`
  the syntax is now
  `setserverheader "XR-Real-IP: %r";`
  This incidentally also makes it possible to change the header name (here: `XR-Real-IP`).

2 Installation for the impatient

For the impatient, here's the very-quick-but-very-superficial recipy for getting crossroads up and running:

- If you don't have SVN or don't want to use it:
  - Obtain the crossroads source archive at `http://crossroads.e-tunity.com`.
  - Change-dir to a 'sources' directory on your system and unpack the archive.
  - Change-dir into the create directory `crossroads/`.
- If you have SVN and want to go for the newest snapshot:
  - Get the latest sources and snapshots using SVN from `svn://svn.e-tunity.com/crossroads`.
  - You'll find the newest alpha version under `crossroads/trunk` and the stable versions under `crossroads/tags`, e.g. `crossroads/tags/release-1.00`.

  - Choose which you want to use: the latest stable release, or the bleeding edge alpha? In the former case, change-dir to `crossroads/tags/release-`*X.YY*, where *X.YY* is a release ID. In the latter case, change-dir to `crossroads/trunk`.
- Type `make install`. This installs the crossroads binary into `/usr/local/bin/`. If the compilation doesn't work on your system, check `etc/Makefile.def` for hints.
- Create a file `/etc/crossroads.conf`. In it state something like:

```
service www {
    port 80;
    revivinginterval 15;
    backend one {
        server 10.1.1.100:80;
    }
    backend two {
        server 10.1.1.101:80;
    }
}
```

  That's off course assuming that you want to balance HTTP on port 80 to two back ends at 10.1.1.100 and 10.1.1.101.
- Type `crossroads start`.
- Surf to the machine where crossroads is running. You will see the pages served by the back ends 10.1.1.100 or 10.1.1.101.
- To monitor the status of crossroads, type `crossroads status`.

3 Using Crossroads

Crossroads is started from the commandline, and highly depends on `/etc/crossroads.conf` (the default configuration file). It supports a number of flags (e.g., to overrule the location of the configuration file). The actual usage information is always obtained by typing `crossroads` without any arguments. Crossroads then displays the allowed arguments.

3.1 General Commandline Syntax

This section shows the most basic usage. As said above, start `crossroads` without arguments to view the full listing of options.

- `crossroads start` and `crossroads stop` are typical actions that are run from system startup scripts. The meaning is self-explanatory.
- `crossroads restart` is a combination of the former two. Beware that a restart may cause discontinuity in service; it is just a shorthand for typing the 'stop' and 'start' actions after one another.
- `crossroad status` reports on each running service. Per service, the state of each back end is reported.
- `crossroads tell` *service backend state* is a command line way of telling crossroads that a given back end, of a given service, is in a given state. Normally crossroads maintains state information itself, but by using `crossroads tell`, a back end can be e.g. taken 'off line' for servicing.
- `crossroads configtest` tells you whether the configuration is syntactially correct.
- `crossroads services` reports on the configured services. In contrast to `crossroads status`, this option only shows what's configured – not what's up and running. Therefore, `crossroads services` doesn't report on back end states.

- `crossroads sampleconf` shows a sample configuration on screen. A good way of quicky viewing the configuration file syntax, or of getting a start for your own configuration `/etc/crossroads.conf`.

3.2 Status Reporting

The command `crossroads status` shows a verbose human-readable report of how Crossroads is doing. When many services are configured, this can be a somewhat lengthy output. If you're interested in the overview of only one service, you can use `crossroads status` *servicename*, in which case the report will only be shown for the stated service.

Similarly, if you're interested only in the status of a given back end of a given service, use `crossroads service` *servicename backendname*.

The flag `-t` causes the status overview to be abbreviated and displayed in a parseable format. This flag can be used in automated scripts that check how Crossroads is doing; e.g., in health checking scripts. When `-t` is used, the format of the status reporter is as follows:

- Service health is reported on one line per service.
- The first string on the line is the service name.
- After this, a series follows of *backendname=state*, where the back end name is the configured name of the back end, and where state is e.g. `available`, `DOWN`, `UNAVAILABLE`, `WAKING`. All statuses that cause Crossroads not to use the back end are in caps. The series is repeated for all back ends of the given service.

3.3 Logging-related options

Two 'flags' of Crossroads are specifically logging-related. This section elaborates on these flags.

First, there's flag `-a`. When present, the start and end of activity is logged using statements like

YYYY-MM-DD HH/MM/SS starting http from 61.45.32.189 to 10.1.1.1

Similarly, there are 'ending' statements. Using this flag and scanning your logs for these statements may be helpful in quickly determining your system load.

Second, there's flag `-l`. This flag selects the 'facility' of logging and defaults to `LOG_DAEMON`. You can supply a number between 0 and 7 to flag `-l` to select `LOG_LOCAL0` to `LOG_LOCAL7`. This would separate the Crossroads-related logging from other streams. Here's a very short guide; please read your Unix manpages of `syslogd` for more information.

- First edit `/etc/syslog.conf` and add a line:

  ```
  local7.*    /var/log/crossroads.log
  ```

  That instructs `syslogd` to send `LOG_LOCAL7` requests to the logfile `/var/log/crossroads.log`.
- Next, restart `syslogd`. On most Unices that's done by issuing `killall -1 syslogd`. (As a side-note, I tried this once on an Bull/AIX system, and the box just shut down. The `killall` command killed every process...)
- Now start `crossroads` with the flag `-l7`.
- Finally, monitor `/var/log/crossroads.log` for Crossroads' messages.

3.4 Reloading Configurations

Crossroads doesn't support the reloading of a configuration while running (such as other programs, e.g. Apache do). There are various technical reasons for this.

However, external lists of allowed or denied IP addresses can be reloaded by sending a signal -1 (`SIGHUP`) to Crossroads. See section 4.2 for the details.

4 The configuration

The configuration that crossroads uses is normally stored in the file `/etc/crossroads.conf`. This location can be overruled using the command line flag `-c`.

This section explains the syntax of the configuration file, and what all settings do.

4.1 General language elements

This section describes the general elements of the crossroads configuration language.

4.1.1 Empty lines and comments

Empty lines are of course allowed in the configuration. Crossroads recognizes three formats of comment:

- C-style, between `/*` and `*/`,
- C++-style, starting with `//` and ending with the end of the text line;
- Shell-style, starting with `#` and ending with the end of the text line.

Simply choose your favorite editor and use the comment that 'looks best'.[3]

4.1.2 Keywords, numbers, identifiers, generic strings

In a configuration file, statements are identified by *keywords*, such as `service`, `verbosity`. These are reserved words.

Many keywords require an *identifier* as the argument. E.g, a service has a unique name, which must start with a letter or underscore, followed by zero or more letters, underscores, or digits. Therefore, in the statement `service myservice`, the keyword is `service` and the identifier is `myservice`.

Other keywords require a numeric argument. Crossroads knows only non-negative integer numbers, as in `port 8000`. Here, `port` is the keyword and `8000` is the number.

Yet other keywords require 'generic strings', such as hostname specifications or system commands. Such generic strings contain any characters (including white space) up to the terminating statement character `;`. If a string must contain a semicolon, then it must be enclosed in single or double quotes:

- `This is a string;` is a string that starts at `T` and ends with `g`
- `"This is a string";` is the same, the double quotes are not necessary
- `"This is ; a string";` has double quotes to protect the inner `;`

Finally, an argument can be a 'boolean' value. Crossroads knows `true`, `false`, `yes`, `no`, `on`, `off`. The keywords `true`, `yes` and `on` all mean the same and can be used interchangeably; as can the keywords `false`, `no` and `off`.

4.2 Service definitions

Service definitions are blocks in the configuration file that state what is for each service. A service definition starts with `service`, followed by a unique identifier, and by statements in `{` and `}`. For example:

---

[3] I favor C or C++ comment. My favorite editor *emacs* can be put in `cmode` and nicely highlight what's comment and what's not. And as a bonus it will auto-indent the configuration!

```
// Definition of service 'www':
service www {
    ...
    ... // statements that define the
    ... // service named 'www'
    ...
}
```

The configuration file can contain many service blocks, as long as the identifying names differ. The following list shows possible statements. Each statement must end with a semicolon, except for the `backend` statement, which has is own block (more on this later).

4.2.1 type - Defining the service type

Description: The `type` statement defines how crossroads handles the stated service. There are currently two types: `any` and `http`. The type `any` means that crossroads doesn't interpret the contents of a TCP stream, but only distributes streams over back ends. The type `http` means that crossroads has to analyze what's in the messages, does magical HTTP header tricks, and so on – all to ensure that multiple connections are treated as one session, or that the back end is notified of the client's IP address.

Unless you really need such special features, use the type `any` (the default), even for HTTP protocols.

Syntax: `type` *specifier*, where *specifier* is `any` or `http`

Default: `any`

4.2.2 port - Specifying the listen port

Description: The `port` statement defines to which TCP port a service 'listens'. E.g. `port 8000` says that this service will accept connections on port 8000.

Syntax: `port` *number*

Default: There is no default. This is a required setting.

4.2.3 bindto - Binding to a specific IP address

Description: The `bindto` statement is used in situations where crossroads should only listen to the stated port at a given IP address. E.g., `bindto 127.0.0.1` causes crossroads to 'bind' the service only to the local IP address. Network connections from other hosts won't be serviced. By default, crossroads binds a service to all presently active IP addresses at the invoking host.

Syntax: `bindto` *address*, where *address* is a numeric IP address, such as 127.0.0.1, or the keyword `any`.

Default: `any`

4.2.4 verbosity - Controlling debug output

Description: Verbosity statements come in two forms: `verbosity on` or `verbosity off`. When 'on', log messages to `/var/log/messages` are generated that show what's going on.[4] The keyword `verbose` is an alias for `verbosity`.

---

[4] Actually, the messages go to `syslog(3)`, using facility `LOG_DAEMON` and priority `LOG_INFO`. In most (Linux) cases this will mean: output to `/var/log/messages`. On Mac OSX the messages go to `/var/log/system.log`.

Syntax: `verbosity` *setting* or `verbose` *setting*, where *setting* is `true`, `yes` or `on` to turn verbosity on; or `false`, `no`, `off` to turn it off.

Default: `off`

4.2.5 dispatchmode - How are back ends selected

Description: The dispatch mode controls how crossroads selects a back end from a list of active back ends. The below text shows the bare syntax. See section 5.1 for a textual explanation.

The settings can be:

- `dispatchmode roundrobin`: Simply the 'next in line' is chosen. E.g, when 3 back ends are active, then the usage series is 1, 2, 3, 1, 2, 3, and so on.
  Roundrobin dispatching is the default method, when no `dispatchmode` statement occurs.
- `dispatchmode random`: Random selection. Probably only for stress testing, though when used with weights (see below) it is a good distributor of new connections too.
- `dispatchmode bysize [ over` *connections* `]`: The next back end is the one that has transferred the least number of bytes. This selection mechanism assumes that the more bytes, the heavier the load.
  The modifier `over` *connections* is optional. (The square brackets shown above are not part of the statement but indicate optionality.) When given, the load is computed as an average of the last stated number of connections. When this modifier is absent, then the load is computed over all connections since startup.
- `dispatchmode byduration [ over` *connections* `]`: The next back end is the one that served connections for the shortest time. This mechanism assumes that the longer the connection, the heavier the load.
- `dispatchmode byconnections`: The next back end is the one with the least active connections. This mechanism assumes that each connection to a back end represents load. It is usable for e.g. database connections.
- `dispatchmode byorder`: The first back end is selected every time, unless it's unavailable. In that case the second is taken, and so on.
- `dispatchmode byclientip`: The client's IP address is 'hashed' into a number, which is used to pick a back end. The same client IP address will therefore always be dispatched to the same back end. When the back end of choice is down, `dispatchmode byconnections` is used.
- `dispatchmode externalhandler` *program arguments*: This is a special mode, where an external program is delegated the responsibility to say which back end should be used next. In this case, Crossroads will call the external program, and this will of course be slower than one of the 'built-in' dispatch modes. However, this is the ultimate escape when custom-made dispatch modes are needed.
  The dispatch mode that uses an `externalhandler` is discussed separately in section 5.3.

The selection algorithm is only used when clients are serviced that aren't part of a sticky HTTP session. This is the case during:

- all client requests of a service type `any`;
- new sessions of a service type `http`.

When type `http` is in effect and a session is underway, then the previously used back end is always selected – regardless of dispatching mode.

Your 'right' dispatch mode will depend on the type of service. Given the fact that crossroads doesn't know (and doesn't care) how to estimate load from a network traffic stream, you have to choose an appropriate dispatch mode to optimize load balancing. In most cases, `roundrobin` or `byconnections` will do the job just fine.

Syntax: `dispatchmode` *mode* (see above for the modes), optionally followed by `over` *number*, or when the *mode* is `externalhandler`, followed by *program*.

Default: `roundrobin`

4.2.6 revivinginterval - Back end wakeup calls

Description: A reviving interval definition is needed when crossroads determines that a back end is temporarily unavailable. This will happen when:

- The back end cannot be reached (network connection fails);
- The network connection to the back end suddenly dies.

An example of the definition is `revivinginterval 10`. When this reviving interval is given, crossroads will check each 10 seconds whether unavailable back ends have woken up yet. A back end is considered awake when a network connection to that back end can succesfully be established.

Syntax: `revivinginterval` *number*, where the number is the interval in seconds.

Default: 0 (no wakeup calls)

4.2.7 maxconnections - Limiting concurrent clients at service level

Description: The maximum number of connections is specified using `maxconnections`. There is one argument; the number of concurrent established connections that may be active within one service.

'Throttling' the number of connections is a way of preventing Denial of Service (DOS) attacks. Without a limit, numerous network connections may spawn so many server instances, that the service ultimately breaks down and becomes unavailable.

Syntax: `maxconnections` *number*, where the number specifies the maximum of concurrent connections to the service.

Default: 0, meaning that all connections will be accepted.

4.2.8 backlog - The TCP Back Log size

Description: The TCP back log size is a number that controls how many 'waiting' network connections may be queued, before a client simply cannot connect. The syntax is e.g. `backlog 5` to cause crossroads to have 5 waiting connections for 1 active connection. The backlog queue shouldn't be too high, or clients will experience timeouts before they can actually connect. The queue shouldn't be too small either, because clients would be simply rejected. Your mileage may vary.

Syntax: `backlog` *number*

Default: 0, which takes the operating system's default value for socket back log size.

4.2.9 shmkey - Shared Memory Access

Description: Different Crossroads invocations must 'know' of each others activity. E.g, `crossroad status` must be able to get to the actual state information of all running services. This is internally implemented through shared memory, which is reserved using a key.

Normally crossroads will supply a shared memory key, based on the service port and bitwise or-ed with a magic number. In situations where this conflicts with existing keys (of other programs, having their own keys), you may supply a chosen value.

The actual key value doesn't matter much, as long as it's unique and as long as each invocation of crossroads uses it.

Syntax: `shmkey` *number*

Default: 0, which means that crossroads will 'guess' its own key, based on TCP port and a magic number.

4.2.10 allow* and deny* - Allowing or denying connections

Description: Crossroads can allow or deny connections based on the IP address of a client. There are four directives that are relevant: `allowfrom`, `allowfile`, `denyfrom` and `denyfile`. When using `allowfrom` and `denyfrom` then the IP addresses to allow or deny connections are stated in `/etc/crossroads.conf`.

When `allow*` directives are used, then all connections are denied unless they match the stated allowed IP's. When `deny*` directives are used, then all connections are allowed unless they match the stated disallowed IP's. When denying and allowing is both used, then the Crossroads checks the deny list first.

The statements `allowfrom` and `denyfrom` are followed by a list of filter specifications. The statements `allowfile` and `denyfile` are followed by a filename; Crossroads will read filter specifications from those external files. In both cases, Crossroads obtains filter specifications and places them in its lists of allowed or denied IP addresses. The difference between specifying filters in `/etc/crossroads.conf` or in external files, is that Crossroads will reload the external files when it receives signal 1 (`SIGHUP`), as in `killall -1 crossroads`.

The filter specifications must obey the following syntax: it consists of up to four numbers ranging from 0 to 255 and separated by a decimal sign. Optionally a slash follows, with a bitmask which is also a decimal number.

This is probably best explained by a few examples:

- `allowfrom 10/8;` will allow connections from `10.*.*.*` (a full Class A network). The mask `/8` means that the first 8 bits of the number (ie., only the `10`) are significant. On the last 3 positions of the IP address, all numbers are allowed. Given this directive, client connections from e.g. 10.1.1.1 and 10.2.3.4 will be allowed.
- `allowfrom 10.3/16;` will allow all IP addresses that start with `10.3`.
- `allowfrom 10.3.1/16;` is the same as above. The third byte of the IP address is superfluous because the netmask specifies that only the first 16 bits (2 numbers) are taken into account.
- `allowfrom 10.3.1.15;` allows traffic from only the specified IP address. There is no bitmask; all four numbers are relevant.
- `allowfrom 10.3.1.15 10.2/16;` allows traffic from one IP address `10.3.1.15` or from a complete Class B network `10.2.*.*`
- `allowfile /tmp/myfile.txt;` in combination with a file `/tmp/myfile.txt`, with the contents `10.3.1.15 10.2/16`, is the same as above.

Syntax:
- `allowfrom` *filter-specificication(s)*
- `denyfrom` *filter-specificication(s)*
- `allowfile` *filename*
- `denyfile` *filename*

Default: In absence of these statements, all client IP's are accepted.

4.2.11 useraccount - Limiting the effective ID of external processes

Description: Using the directive `useraccount`, the effective user and group ID can be restricted. This comes into effect when Crossroads runs external commands, such as:

- Hooks for `onstart`, `onend` or `onfail`;
- External dispatchers, when `dispatchmode externalhandler` is in effect.

Once a user name for external commands is specified, Crossroads assumes the associated user ID and group ID before running those commands.

Syntax: `useraccount` *username*

Default: None; when unspecified, external commands are run with the ID that was in effect when Crossroads was started.

4.3 Backend definitions

Inside the service definitions as are described in the previous section, *backend definitions* must also occur. Backend definitions are started by the keyword `backend`, followed by an identifier (the back end name) , and statements inside `{` and `}`:

```
service myservice {
    ...
    ... // statements that define the
    ... // service named 'myservice'
    ...

    backend mybackend {
        ...
        ... // statements that define the
        ... // backend named 'mybackend'
        ...
    }
}
```

Each service definition must have at least one backend definition. There may be more (and probably will, if you want balancing and fail over) as long as the backend names differ. The statements in the backend definition blocks are described in the following sections.

Some directives (`stickycookie` etc.) only have effect when Crossroads treats the network traffic as a stream of HTTP messages; i.e., when the service is declared with `type http`. Incase of `type any`, the HTTP-specific directives have no effect.

4.3.1 server - Specifying the back end address

Description: Each back end must be identified by the network name (server name) where it is located. For example: `server 10.1.1.23`, or `server web.mydomain.org`. A TCP port specifier can follow the server name, as in `server web.mydomain.org:80`.

Syntax:
- `server` *servername*, where *servername* is a network name or IP address;
- `server` *servername:port*

Default: There is no default. This is a required setting.

4.3.2 verbosity - Controlling verbosity at the back end level

Description: Similar to `service` specifications, a `backend` can have its own verbosity (`on` or `off`). When `on`, traffic to and fro this back end is reported.

Syntax:
- `verbosity` *setting*, or
- `verbose` *setting*, where *setting* is `true`, `yes` or `on`, or `false`, `no`, `off` to turn it off.

Default: `off`

4.3.3 retries - Specifying allowed failures

Description: Back ends that are ’flaky’ or on a less reliable network can be marked as unavailable after not just one failure, but after e.g. three. You can use this configuration if you suspect that spurious errors cause otherwise ’good’ back ends to be marked as unavailable, while they in fact still could be used.

Syntax: `retries` *number*; where *number* is the threshold of bad connections. Once exceeded, Crossroads will mark a back end as unavailable.

Default: 1; a back end is assumed to be unavailable after the first bad connection.

4.3.4 weight - When a back end is more equal than others

Description: To influence how backends are selected, a backend can specify its ’weight’ in the process. The higher the weight, the less likely a back end will be chosen. The default is 1.

The weighing mechanism only applies to the dispatch modes `random`, `byconnections`, `bysize` and `byduration`. The weight is in fact a penalty factor. E.g., if backend A has `weight 2` and backend B has `weight 1`, then backend B will be selected all the time, until its usage parameter is twice as large as the parameter of A. Think of it as a ’sluggishness’ statement.

Syntax: `weight` *number*; the higher the number, the more ’sluggish’ a back end is

Default: 1; all back ends have equal weight.

4.3.5 decay - Levelling out activity of a back end

Description: To make sure that a ’spike’ of activity doesn’t influence the perceived load of a back end forever, you may specify a certain decay. E.g, the statement `decay 10` makes sure that the load that crossroads computes for this back end (be it in seconds or in bytes) is decreased by 10% each time that **an other** back end is hit. Decays are not applied to the count of concurrent connections.

This means that when a given back end is hit, then its usage data of the transferred bytes and the connection duration are updated using the actual number of bytes and actual duration. However, when a different back end is hit, then the usage data are decreased by the specified decay.

Syntax: `decay` *number*, where *number* is a percentage that decreases the back end usage data when other back ends are hit.

Default: 0, meaning that no decay is applied to usage statistics.

4.3.6 onstart, onend, onfail - Action Hooks

Description: The three directives `onstart`, `onend` and `onfail` can be specified to start system commands (external programs) when a connection to a back end starts, fails or ends:

- `onstart` commands will be run when Crossroads successfully connects to a back end, and starts servicing;
- `onend` commands will be run when a (previously established) connection stops;
- `onfail` commands will be run when Crossroads tries to contact a back end to serve a client, but the back end can’t be reached.

The format is always `on`*type command*. The *command* is an external program, optionally followed by arguments. The command is expanded according to the following table:

- `%a` is the availability of the current back end, when a current back end is established;

- `%1a` is the availability of the first back end (0 when unavailable, 1 if available); `%2a` is the availability of the second back end, and so on;
- `%b` is the name of the current back end, when one is established;
- `%1b` is the name of the first back end, `%2b` of the second back end, and so on;
- `%e` is the count of seconds since start of epoch (January 1st 1970 GMT);
- `%r` is the IP address of the client that requests a connection and for whom the external dispatcher should compute a back end;
- `%s` is the name of the current service that the client connected to;
- `%t` is the current local time in ASCII format, in *YYYY-MM-DD/hhh:mm:ss*;
- `%T` is the current GMT time in ASCIII format;
- `%v` is the Crossroads version;
- Any other chararacter following a `%` sign is taken literally; e.g. `%z` is just a z.

Syntax:
- `onstart` *commandline*
- `onend` *commandline*
- `onfail` *commandline*
- `onsuccess` *commandline*

Default: There is no default. Normally no external programs are run upon connection, success or failure of a back end.

4.3.7 trafficlog and throughputlog - Debugging and Performance Aids

Description: Two directives are available to log network traffic to files. They are `trafficlog` and `throughputlog`.

The `trafficlog` statement causes all traffic to be logged in hexadecimal format. Each line is prefixed by `B` or `C`, depending on whether the information was received from the back end or from the client.

The `throughputlog` statement writes shorthand transmissions to its log, accompanied by timings.

Syntax:
- `trafficlog` *filename*
- `throughputlog` *filename*

Default: none

4.3.8 stickycookie - Back end selection with an HTTP cookie

Description: The directive `stickycookie` *value* causes Crossroads to unpack clients' requests, to check for *value* in the cookies. When found, the message is routed to the back end having the appropriate `stickycookie` directive.

E.g., consider the following configuration:

```
service ... {
    ...
    backend one {
        ...
        stickycookie "BalancerID=first";
    }
    backend two {
        ...
        stickycookie "BalancerID=second";
    }
}
```

When clients' messages contain cookies named `BalancerID` with the value `first`, then such messages are routed to backend `one`. When the value is `second` then they are routed to the backend `two`.

There are basically to provide such cookies to a browser. First, a back end can insert such a cookie into the HTTP response. E.g., the webserver of back end `one` might insert a cookie named `BalancerID`, having value `first`. Second, Crossroads can insert such cookies using a carefully crafted directive `addclientheader`.

Syntax: `stickycookie` *cookievalue*

Default: There is no default.

4.3.9 HTTP Header Modification Directives

Description: Crossroads understands the following header modification directives: `addclientheader`, `appendclientheader`, `setclientheader`, `addserverheader`, `appendserverheader`, `setserverheader`.

The directive names always consist of *ActionDestination*`header`, where:

- The action is `add`, `append` or `insert`.
  - Action `add` adds a header, even when headers with the same name already are present in an HTTP message. Adding headers is useful for e.g. `Set-Cookie` headers; a message may contain several of such headers.
  - Action `append` adds a header if it isn't present yet in an HTTP message. If such a header is already present, then the value is appended to the pre-existing header. This is useful for e.g. `Via` headers. Imagine an HTTP message with a header `Via: someproxy`. Then the directive `appendclientheader "Via: crossroads"` will rewrite the header to `Via: someproxy; crossroads`.
  - Action `set` overwrites headers with the same name; or adds a new header if no pre-existing is found. This is useful for e.g. `Host` headers.
- The destination is one of `client` or `server`. When the destination is `server`, then Crossroads will apply such directives to HTTP messages that originate from the browser and are being forwarded to back ends. When the destination is `client`, then Crossroads will apply such directives to backend responses that are shuttled to the browser.

The format of the directives is e.g. `addclientheader "X-Processed-By: Crossroads"`. The directives expect one argument; a string, consisting of a header name, a colon, and a header value. As usual, the directive must end with a semicolon.

The header value may contain one of the following formatting directives:

- `%a` is the availability of the current back end, when a current back end is established;
- `%1a` is the availability of the first back end (0 when unavailable, 1 if available); `%2a` is the availability of the second back end, and so on;
- `%b` is the name of the current back end, when one is established;
- `%1b` is the name of the first back end, `%2b` of the second back end, and so on;
- `%e` is the count of seconds since start of epoch (January 1st 1970 GMT);
- `%r` is the IP address of the client that requests a connection and for whom the external dispatcher should compute a back end;
- `%s` is the name of the current service that the client connected to;
- `%t` is the current local time in ASCII format, in *YYYY-MM-DD/hhh:mm:ss*;
- `%T` is the current GMT time in ASCIII format;
- `%v` is the Crossroads version;
- Any other chararacter following a `%` sign is taken literally; e.g. `%z` is just a z.

The following examples show common uses of header modifications.

Enforcing session stickiness: By combining `stickycookie` and `addclientheader`, HTTP session stickiness is enforced. Consider the following configuration:

```
service ... {
    ...
    backend one {
        ...
        addclientheader "Set-Cookie: BalancerID=first; path=/";
        stickycookie "BalancerID=first";
    }
    backend two {
        ...
        addclientheader "Set-Cookie: BalancerID=second; path=/";
        stickycookie "BalancerID=second";
    }
}
```

The first request of an HTTP session is balanced to either backend `one` or `two`. The server response is enriched using `addclientheader` with an appropriate cookie. A subsequent request from the same browser now has that cookie in place; and is therefore sent to the same back end where the its predecessors went.

Hiding the server software version: Many servers (e.g. Apache) advertize their version, as in `Server: Apache 1.27`. This potentially provides information to attackers. The following configuration hides such information:

```
service ... {
    ...
    backend one {
        ...
        setclientheader "Server: WWW-Server";
    }
}
```

Informing the server of the clients' IP address: Since Crossroads sits 'in the middle' between a client and a back end, the back end perceives Crossroads as its client. The following sends the true clients' IP address to the server, in a header `X-Real-IP`:

```
service ... {
    ...
    backend one {
        ...
        setserverheader "X-Real-IP: %r";
    }
}
```

Keep-Alive Downgrading: The directives `setclientheader` and `setserverheader` also play a key role in downgrading Keep-Alive connections to 'single-shot'. E.g., the following configuration makes sure that no Keep-Alive connections occur.

```
service ... {
    ...
    backend one {
        ...
        setserverheader "Connection: close";
        setclientheader "Connection: close";
    }
}
```

Syntax:

- `addclientheader` *Headername: headervalue* to add a header in the traffic towards the client, even when another header *Headername* exists;

- `appendclientheader` *Headername: headervalue* to append *headervalue* to an existing header *Headername* in the traffic towards the client, or to add the whole header alltogether;
- `setclientheader` *Headername: headervalue* to overwrite an existing header in the traffic towards the client, or to add such a header;
- `addserverheader` *Headername: headervalue* to add a header in the traffic towards the server, even when another header *Headername* exists;
- `appendserverheader` *Headername: headervalue* to append *headervalue* to an existing header *Headername* in the traffic towards the server, or to add the whole header alltogether;
- `setserverheader` *Headername: headervalue* to overwrite an existing header in the traffic towards the server, or to add such a header.

Default: There is no default.

5 Tips, Tricks and Random Remarks

The following sections elaborate on the directives as described in section 4 to illustrate how crossroads works and to help you achieve the "optimal" balancing configuration.

5.1 How back ends are selected in load balancing

In order to tune your load balancing, you'll need to understand how crossroads computes usage, how weighing works, and so on. In this section we'll focus on the dispatching modes `bysize`, `byduration` and `byconnections` only. The other dispatching types are self-explanatory.

5.1.1 Bysize, byduration or byconnections?

As stated before, crossroads doesn't know 'what a service does' and how to judge whether a given back end is very busy or not. You must therefore give the right hints:

- In general, a service which is CPU bound, will be more busy when it takes longer to process a request. The dispatch mode `byduration` is appropriate here.
- In contrast, a service which is filesystem bound, will be more busy when more data are transferred. The dispatch mode `bysize` is appropriate.
- The dispatch mode `byduration` can also be used when network latency is an issue. E.g., if your balancer has back ends that are geograpically distributed, then `byduration` would be a good way to select best available back ends.
- Furthermore it is noteworthy that `dispatchmode byduration` is not usable for interactive processes such as SSH logins. Idle time of a login adds to the duration, while causing (almost) no load. Mode `byduration` should only be used for automated processes that don't wait for user interaction (e.g., SOAP calls and other HTTP requests).
- As a last remark, the dispatching mode `byconnections` can be used if you don't have other clues for load estimations.

  E.g., consider a database connection. What's heavier on the back end, time-consuming connections, or connections where loads of bytes are transferred? Well, that depends. A tough `select` query that joins multiple tables can be very heavy on the back end, though the response set can be quite small - and hence the number of transferred bytes. That would suggest dispatching by duration. However, `byduration` balancing doesn't respresent the true world, when interactive connections can occur where users have an idle TCP connection to the database: this consumes time, but no bytes (see the SSH login example above). In this case, the dispatch mode `byconnections` may be your best bet.

5.1.2 Averaging size and duration

The configuration statement `dispatchmode bysize` or `byduration` allows an optional modifier `over` *number*, where the stated number represents a connection count. When this modifier is present, then crossroads will use a moving average over the last *n* connections to compute duration and size figures.

In the real world you'll always want this modifier. E.g., consider two back ends that are running for years now, and one of them is suddenly overloaded and very busy (it experiences a 'spike' in activity). When the `over` modifier is absent, then the sudden load will hardly show up in the usage figures – it will flatten out due to the large usage figures already stored in the years of service.

In contrast, when e.g. `over 3` is in effect, then a sudden load does show up – because it highly contributes to the average of three connections.

5.1.3 Specifying decays

Decays are also only relevant when crossroads computes the 'next best back end' by size (bytes) or duration (seconds). E.g., imagine two back ends A and B, both averaged over say 3 connections.

Now when back end A is suddenly hit by a spike, its average would go up accordingly. But the back end would never again be used, unless B also received a similar spike, because A's 'usage data' over its last three connections would forever be larger than B's data.

For that reason, you should in real situations probably always specify a decay, so that the backend selection algorithm recovers from spikes. Note that the usage data of the back end where a decay is specified, decay when **other** back ends are hit. The decay parameter is like specifying how fast your body regenerates when someone else does the work.

The below configuration illustrates this:

```
/* Definition of the service */
service soap {
    /* Local TCP port */
    port 8080;

    /* We'll select back ends by the processing
     * duration
     */
    dispatchmode byduration over 3;

    /* First back end: */
    backend A {
        /* Back end IP address and port */
        server 10.1.1.1:8080;

        /* When this back end is NOT hit because
         * the other one was less busy, then the
         * usage parameters decay 10% per connection
         */
        decay 10;
    }

    /* Second back end: */
    backend B {
        server 10.1.1.2:8080;
        decay 10;
    }
}
```

5.1.4 Adjusting the weights

The back end modifier `weight` is useful in situations where your back ends differ in respect to performance. E.g,. your back ends may be geographically distributed, and you know that a given back end is difficult to reach and often experiences network lag.

Or you may have one primary back end, a system with a fast CPU and enough memory, and a small fall-back back end, with a slow CPU and short on memory. In that case you know in advance that the second back end should be used only rarely. Most requests should go to the big server, up to a certain load.

In such cases you will know in advance that the best performing back ends should be selected the most often. Here's where the `weight` statement comes in: you can simply increase the weight of the back ends with the least performance, so that they are selected less frequently.

E.g., consider the following configuration:

```
service soap {
    port 8080;
    dispatchmode byduration over 3;
    backend A {
        server 10.1.1.1:8080;
        decay 20;
    }
    backend B {
        server 10.1.1.2:8080;
        weight 2;
        decay 10;
    }
    backend C {
        server 10.1.1.3:8080;
        weight 4;
        decay 5;
    }
}
```

This will cause crossroads to select back ends by the processing time, averaging over the last three connections. However, backend B will kick in only when its usage is half of the usage of A (back end B is probably only half as fast as A). Backend C will kick in only when its usage is a quarter of the usage of A, which is half of the usage of B (back end C is probably very weak, and just a fall-back system incase both A and B crash). Note also that A's usage data decay much faster than B's and C's: we're assuming that this big server recovers quicker than its smaller siblings.

5.2 Throttling the number of concurrent connections

If you suspect that your service may occasionally receive 'spikes' of activity[5], then it might be a good idea to protect your service by specifying a maximum number of concurrent connections. This protection can be specified on two levels:

On the service level a statement like `maxconnections 100;` states that the service as a whole will never service more than 100 concurrent connections. This means that all your back ends and the crossroads balancer itself will be protected from being overloaded.

---

[5] which you should always assume

On the back end level a statement like `maxconnections 10;` states that this particular back end will never have more than 10 concurrent connections; regardless of the overall setting on the service level. This means that this particular back end will be protected from being overloaded (regardless of what other back ends may experience).

The `maxconnections` statement, combined with a back end selection algorithm, allows very fine granularity. The `maxconnections` statement on the back end level is like a hand brake: even when you specify a back end algorithm that would protect a given back end from being used too much, a situation may occur where that back end is about to be hit. A `maxconnections` statement on the level of that back may then protect it.

5.3 Using an external program to dispatch

As mentioned before, Crossroads supports several built-in dispatch modes. However, you are always free to hook-in your own dispatch mode that determines the next back end using your own specific algorithm. This section explains how to do it.

5.3.1 Configuring the external handler

First, the `dispatchmode` statement needs to inform Crossroads that an external program will do the job. The syntax is: `dispatchmode externalhandler` *program arguments*. The *program* must point to an executable program that will be started by Crossroads. The specifier *arguments* can be anything you want; those will be the arguments to Crossroads. You can however use the following special format specifiers:

- `%a` is the availability of the current back end, when a current back end is established;
- `%1a` is the availability of the first back end (0 when unavailable, 1 if available); `%2a` is the availability of the second back end, and so on;
- `%b` is the name of the current back end, when one is established;
- `%1b` is the name of the first back end, `%2b` of the second back end, and so on;
- `%e` is the count of seconds since start of epoch (January 1st 1970 GMT);
- `%r` is the IP address of the client that requests a connection and for whom the external dispatcher should compute a back end;
- `%s` is the name of the current service that the client connected to;
- `%t` is the current local time in ASCII format, in *YYYY-MM-DD/hhh:mm:ss*;
- `%T` is the current GMT time in ASCIII format;
- `%v` is the Crossroads version;
- Any other chararacter following a `%` sign is taken literally; e.g. `%z` is just a z.

Note that the format specifiers such as `%b` don't make sense in the phase in which an external handler is called, since there is no current back end yet (the job of the handler is to supply one).

5.3.2 Writing the external handler

The external handler is activated using the arguments that are specified in `/etc/crossroads.conf`. The external handler can do whatever it wants, but ultimately, it must write a back end name on its *stdout*. Crossroads reads this, and if the back end is available, uses that back end for the connection.

5.3.3 Examples of external handlers

This section shows some examples of Crossroads configurations vs. external handlers. The sample handlers that are shown here, are also included in the Crossroads distribution, under the directory `etc/`. Also note that the examples shown here are just quick-and-dirty Perl scripts, meant to illustrate only. Your applications may need other external handlers, but you can use the shown scripts as a starting point.

Round-robin dispatching This example is trivial in the sense that round-robin dispatching is already built into Crossroads, so that using an external handler for this purpose only slows down Crossroads. However, it's a good starting example.

The Crossroads configuration is shown below:

```
service test {
    port 8001;
    verbosity on;
    revivinginterval 5;

    dispatchmode externalhandler
        /usr/local/src/crossroads/etc/dispatcher-roundrobin
            %1b %1a %2b %2a;

    backend testone {
        server localhost:3128;
        verbosity on;
    }
    backend testtwo {
        server locallhost:3128;
        verbosity on;
    }
}
```

The relevant `dispatchmode` statement invokes the external program `dispatcher-roundrobin` with four arguments: the name of the first back end (`testone`), its availability (0 or 1), the name of the second back end (`testtwo`) and its availability (0 or 1).

The external handler, which is also included in the Crossroads distribution, is shown below. It is a Perl script.

```
#!/usr/bin/perl

use strict;

# Example of a round-robin external dispatcher. This is totally
# superfluous, Crossroads has this on-board; if you use the external
# program for determining round-robin dispatching, then you'll only
# slow things down. This script is just meant as an example.

# Globals / configuration
# -----------------------
my $log = '/tmp/exthandler.log';    # Debug log, set to /dev/null to suppress
my $statefile = '/tmp/rr.last';     # Where we keep the last used

# Logging
# -------
sub msg {
    return if ($log eq '/dev/null' or $log eq '');
    open (my $of, ">>$log") or return;
    print $of (scalar(localtime()), ' ', @_);
}

# Read the last used back end
# --------------------------
sub readlast() {
    my $ret;

    if (open (my $if, $statefile)) {
        $ret = <$if>;
        chomp ($ret);
        close ($if);
        msg ("Last used back end: $ret\n");
        return ($ret);
    }
    msg ("No last-used back end (yet)\n");
    return (undef);
}

# Write back the last used back end, reply to Crossroads and stop
# ---------------------------------------------------------------
sub reply ($) {
    my $last = shift;

    if (open (my $of, ">$statefile")) {
        print $of ("$last\n");
    }
    print ("$last\n");
    exit (0);
}

# Main starts here
# ----------------

# Collect the cmdline arguments. We expect pairs of backend-name /
# backend-availablility, and we'll store only the available ones.
msg ("Dispatch request received\n");
my @backend;
for (my $i = 0; $i <= $#ARGV; $i += 2) {
    push (@backend,  $ARGV[$i]) if ($ARGV[$i + 1]);
}
msg ("Available back ends: @backend\n");

# Let's see what the last one is. If none found, then we return the
# first available back end. Otherwise we need to go thru the list of
# back ends, and return the next one in line.
my $last = readlast();
if ($last eq '') {
    msg ("Returning first available back end $backend[0]\n");
    reply ($backend[0]);
}

# There **was** a last back end.  Try to match it in the list,
# then return the next-in-line.
for (my $i = 0; $i < $#backend; $i++) {
    if ($last eq $backend[$i]) {
        msg ("Returning next back end ", $backend[$i + 1], "\n");
        reply ($backend[$i + 1]);
    }
}

# No luck.. run back to the first one.
msg ("Returning first back end $backend[0]\n");
reply ($backend[0]);
```

The working of the script is basically as follows:

- The argument list is scanned. Back ends that are available are collected in an array `@backend`.
- The script queries a state file `/tmp/rr.last`. If a back end name occurs there, then the next back end is looked up in `@backend` and returned to Crossroads. If no last back is unknown or can't be matched, then the first available back end (first element of `@backend`) is returned to Crossroads.
- Informing Crossroads is done via the subroutine `reply()`. This code writes the selected back end to file `/tmp/rr.last` (for future usage) and prints the back end name to *stdout*.
- The script logs its actions to a file `/tmp/exthandler.log`. This log file can be inspected for the script's actions.

Dispatching by the client IP address The following example shows a useful real-life situation that illustrates how dispatching by client IP address works. **Note that** as of Crossroads 1.31, `dispatchmode byclientip` is implemented – so that the below description is somewhat superfluous. The code snippets however can help you in modelling your own specific dispatch modes, aided by external helpers.

The situation is as follows:

- Crossroads is used as a single-address point to forward Remote Desktop requests to a farm of Windows systems, where users can work via remote access;
- However, users may stop their session, and when they re-connect, they expect to be sent to the Windows system that they had worked on previously;
- Client PC's have their distinct IP addresses, which distinguishes them.
- Of four windows systems, two are large servers, and two are small ones. We'll want to assign large servers to clients when we have a choice.

The requirements resemble session stickiness in HTTP, except that the remote desktop protocol doesn't support stickiness. This situation is a perfect example of how an external handler can help:

- A suitable dispatch mode isn't yet available in Crossroads, but can be easily coded in an external handler;
- The potential delay due to the calling of an external handler won't even be noticed. This is a network service where the connection time isn't critical; we'd expect only a few (albeit lengthy) TCP connections.

The approach to the solution of this problem uses several external program hooks:

- An external dispatcher handler will be responsible for suggesting a back end, given a client IP and given the current timestamp. This handler will consult an internal administration to see whether the stated IP address should re-use a back end, or to determine which back end is free for usage.

- An external hook `onstart` will be responsible for updating the internal administration; i.e., to flag a back end as 'occupied'.
- The external hooks `onfailure` and `onend` will be responsible for flagging a back end as 'free' again; i.e., for erasing any previous information that states that the back end was occupied.

The Crossroads configuration is shown below. Only four Windows back ends are shown. Each back end is configured on a given IP address, port 3389, and is limited to one concurrent connection (otherwise a new user might 'steal' a running desktop session).

```
service rdp {
    port 3389;
    revivinginterval 5;

    /* rdp-helper dispatch IP STAMP ...  will suggest a back end to use,
     * arguments are for all back ends: name, availability, weight */
    dispatchmode externalhandler
        /usr/local/src/crossroads/etc/rdp-helper dispatch %r %e
            %1b %1a %1w
            %2b %2a %2w
            %3b %3a %3w
            %4b %4a %4w;

    backend win1 {
        server 10.1.1.1:3389;
        maxconnections 1;
        /* rdp-helper start IP STAMP BACKEND will log the actual start
         * of a connection;
         * rdp-helper end IP will log the ending of a connection */
        onstart /usr/local/src/crossroads/etc/rdp-helper start %r %e %b;
        onend   /usr/local/src/crossroads/etc/rdp-helper end %r;
        onfail  /usr/local/src/crossroads/etc/rdp-helper end %r;
    }
    backend win2 {
        server 10.1.1.2:3389;
        maxconnections 1;
        onstart /usr/local/src/crossroads/etc/rdp-helper start %r %e %b;
        onend   /usr/local/src/crossroads/etc/rdp-helper end %r;
        onfail  /usr/local/src/crossroads/etc/rdp-helper end %r;
    }
    backend win3 {
        server 10.1.1.3:3389;
        maxconnections 1;
        weight 2;
        onstart /usr/local/src/crossroads/etc/rdp-helper start %r %e %b;
        onend   /usr/local/src/crossroads/etc/rdp-helper end %r;
        onfail  /usr/local/src/crossroads/etc/rdp-helper end %r;
    }
    backend win4 {
        server 10.1.1.4:3389;
        maxconnections 1;
        weight 3;
        onstart /usr/local/src/crossroads/etc/rdp-helper start %r %e %b;
        onend   /usr/local/src/crossroads/etc/rdp-helper end %r;
        onfail  /usr/local/src/crossroads/etc/rdp-helper end %r;
    }
}
```

Depending on the dispatcher stage, the exernal handler `rdp-helper` is invoked in different ways:

During dispatching the helper is called to suggest a back end. The arguments are an action indicator `dispatch`, the client's IP address, the timestamp, and four triplets that represent back ends: per back end its name, its availability, and its weight. The purpose of the helper is to tell Crossroads which back end to use.

During connection start the helper will be invoked to inform it of the start of a connection, given a client IP address.

When a connection terminates the helper will be invoked to inform it that the connection has ended.

Here's the external handler as Perl script. It uses the module `GDBM_File` which most likely will not be part of standard Perl distributions, but can be added using CPAN. (Alternatively, any other database module can be used.)

```
#!/usr/bin/perl

use strict;
use GDBM_File;

# Global variables and configuration
# ------------------------------------
my $log = '/tmp/exthandler.log';    # Debug log, set to /dev/null to suppress
my $cdb = '/tmp/client.db';         # GDBM database of clients
my %db;                             # .. and memory representation of it
my $timeout = 24*60*60;             # Timeout of a connection in secs

# Logging
# -------
sub msg {
    return if ($log eq '/dev/null' or $log eq '');
    open (my $of, ">>$log") or return;
    print $of (scalar(localtime()), ' ', @_);
    close ($of);
}

# Reply a back end to the caller and stop processing.
# ----------------------------------------------------
sub reply ($) {
    my $b = shift;
    msg ("Suggesting $b to Crossroads.\n");
    print ("$b\n");
    exit (0);
}

# Is a value in an array
# ----------------------
sub inarray {
    my $val = shift;
    for my $other (@_) {
        return (1) if ($other eq $val);
    }
    return (0);
}

# A connection is starting
# ------------------------
sub start {
    my ($ip, $stamp, $backend) = @_;
    msg ("Logging START of connection for IP $ip on stamp $stamp, ",
         "back end $backend\n");
    $db{$ip} = "$backend:$stamp";
}

# A connection has ended
# ----------------------
sub end {
    my $ip = shift;
    msg ("Logging END of connection for IP $ip\n");
    $db{$ip} = undef;
}

# Request to determine a back end
# -------------------------------
sub dispatch {
    my $ip = shift;
    my $stamp = shift;

    msg ("Request to dispatch IP $ip on stamp $stamp\n");

    # Read the next arguments. They are triplets of
    # backend-name / availability / weight. Store if the back end is
    # available.
    my (@backends, @weights);
    for (my $i = 0; $i < $#_; $i += 3) {
        if ($_[$i + 1] != 0) {
            push (@backends, $_[$i]);
            push (@weights,  $_[$i + 2]);
            msg ("Candidate back end: $_[$i] with weight ", $_[$i + 2], "\n");
        }
    }

    # See if this is a reconnect by a previously seen client IP. We’ll
    # treat this as a reconnect if the timeout wasn’t yet exceeded.
    if ($db{$ip} ne ’’) {
        my ($last_backend, $last_stamp) = split (/:/, $db{$ip});
        msg ("IP $ip had last connected on $last_stamp to $last_backend\n");
        if ($stamp < $last_stamp + $timeout) {
            msg ("Timeout not yet exceeded, this may be a reconnect\n");
            # We’ll allow a reconnect only if the stated last_backend is
            # free (sanity check).
            if (inarray ($last_backend, @backends)) {
                msg ("Last back end $last_backend is available, ",
                     "letting through\n");
                reply ($last_backend);
            } else {
                msg ("Last used back end isn’t free, suggesting a new one\n");
            }
        } else {
            msg ("Timeout exceeded, suggesting a new back end\n");
        }
    } else {
        msg ("Np preveious connection data, suggesting a new back end\n");
    }

    my $bestweight = -1;
    my $bestbackend;
    for (my $i = 0; $i <= $#weights; $i++) {
        if ($bestweight == -1 or $bestweight > $weights[$i]) {
            $bestweight  = $weights[$i];
            $bestbackend = $backends[$i];
        }
    }

    msg ("Best back end: $bestbackend (given weight $bestweight)\n");
    reply ($bestbackend);
}

# Main starts here
# ----------------
msg ("Start of run, attaching GDBM database ’$cdb’\n");
tie (%db, ’GDBM_File’, $cdb, &GDBM_WRCREAT, 0600);

# The first argument must be an action ’dispatch’, ’start’ or ’end’.
# Depending on the action, we do stuff.
my $action = shift (@ARGV);
if ($action eq ’dispatch’) {
    dispatch (@ARGV);
} elsif ($action eq ’start’) {
    start (@ARGV);
} elsif ($action eq ’end’) {
    end (@ARGV);
} else {
    print STDERR ("Usage: rdp-helper {dispatch|start|end} args\n");
    exit (1);
}
```

5.4 TCP Session Stickiness

If you need to make sure that a client that once gets dispatched to a given back end keeps re-visiting the back end, then Crossroads offers the dispatch mode `byclientip`. This mode will only work when each client is seen by Crossroads with its own specific IP address; ie., this method won’t work when clients reach Crossroads through a masquerading firewall (in which case all clients would be seen as having the firewall’s IP address).

The `dispatchmode byclientip` works as follows:

- The client’s IP address is taken in its string representation and ’hashed’ into a number.
- The number is brought back to the number of available back ends (using a modulo-operation).

- The result defines the back end of choice.

If the preferred back end is unavailable, then the action that Crossroads takes is to dispatch as if `byconnections`: of the available back ends, the one with the least connections is taken.

5.5 HTTP Session Stickiness

This section focuses on HTTP session stickiness. This term refers to the ability of a balancer to route a conversation between browser and a backend farm with webservers always to the same back end. In other words: once a back end is selected by the balancer, it will remain the back end of choice, even for subsequent connections.

5.5.1 Don't use stickiness!

The rule of thumb as far as the balancer is concerned, is: **Do not use HTTP session stickiness unless you really have to.** Enabling session stickiness hampers failover, balancing and performance:

- Failover is hampered because during the session, the balancer has to assign new connections to the same back end that was selected at the start of a session. If the back end suddenly goes 'down', then the session will most likely crash. (Actually, when a back end becomes unreachable in the middle of a session, Crossroads will assign a new back end to that session. This will most likely result in a malfunction of the underlying application.)
- Balancing is hampered because at the start of the session, the balancer has selected the next-best back end. But during the session, that back end may well become overloaded. The balancer however must continue to send the requests there.
- Performance is hampered because crossroads needs to 'unpack' messages as they are passed to and fro. That's because crossroads needs to check the HTTP headers in the messages for persistence cookies.

There is a number of measures that you can take to avoid using session stickiness. E.g., session data can be 'shared' between web back ends. PHP offers functionality to store session data in a database, so that all PHP applications have access to these data. Application servers such as Websphere can be configured to replicate session data between nodes.

5.5.2 But if you must..

If you really need stickiness, think first whether you might use TCP stickiness (using the client's IP address to dispatch). If you can, then this is the preferred method, since Crossroads won't have to unpack TCP streams. Below is a short configuration example:

```
service www {
    port 80;
    type any;
    revivinginterval 15;
    dispatchmode byclientip;

    backend one {
        server 10.1.1.100:80;
    }

    backend two {
        server 10.1.1.101:80;
    }
}
```

However, if you **must** use HTTP-base session stickiness, then proceed as follows:

- At the level of a `service` description, set the type to `http`.
- At the level of each back end description, configure the `stickycookie` and a `addclientheader` directives.

Once crossroads sees that, it will examine each HTTP message that it shuttles between client and back end:

- If there is no persistence cookie in the HTTP headers of a client's request, then the message must be the first one and a new session should be established. Crossroads selects an appropriate back end, sends the message to that back end, catches the reply, and inserts a `Set-Cookie` directive.
- If there is a persistence cookie in the HTTP headers of a client's request, then the request is part of an already established session. Crossroads analyzes the cookie and forwards the request to the appropriate back end.

Below is a short example of a configuration.

```
service www {
    port 80;
    type http;
    revivinginterval 15;
    dispatchmode byconnections;

    backend one {
        server 10.1.1.100:80;
        stickycookie XRID=100;
        addclientheader "Set-Cookie: XRID=100; Path=/";
    }

    backend two {
        server 10.1.1.101:80;
        stickycookie XRID=101;
        addclientheader "Set-Cookie: XRID=101; Path=/";
    }
}
```

Note how the cookie names and values in the directives `stickycookie` and `addclientheader` match. That is obviously a prerequisite for stickiness.

5.6 Passing the client's IP address

Since Crossroads just shuttles bytes to and fro, meta-information of network connections is lost. As far as the back ends are concerned, their connections originate at the Crossroads junction. For example, standard Apache access logs will show the IP address of Crossroads.

In order to compensate for this, Crossroads can insert a special header in HTTP connections, to inform the back end of the original client's IP address. In order to enable this, the Crossroads configuration must state the following:

- The service type must be `http`, and not `any`;
- In the back end definition, the following statement must occur:
  `addserverheader "X-Real-IP: %r";`
  You are of course free to choose the header name; the here used `X-Real-IP` is a common name for this purpose.

After this, HTTP traffic that arrives at the back ends has a new header: `X-Real-IP`, holding the client's IP address. **Note that** once the type is set to `http`, Crossroads' performance will be hampered – all passing messages will have to be unpacked and analyzed.

5.6.1 Sample Crossroads configuration

The below sample configuration shows two HTTP back ends that receive the client's IP address:

```
service www {
    port 80;
    type http;
    revivinginterval 5;
    dispatchmode roundrobin;

    backend one {
        server 10.1.1.100:80;
        addserverheader "X-Real-IP: %r";
    }

    backend two {
        server 10.1.1.200:80;
        addserverheader "X-Real-IP: %r";
    }
}
```

5.6.2 Sample Apache configuration

The method by which each back end analyzes the header `X-Real-IP` will obviously be different per server implementations. However, a common method with the Apache webserver is to log the client's IP address into the access log.

Often this is accomplished using the log format `custom`, defined as follows:

```
LogFormat "%h %l %u %t %D \"%r\" %>s %b" common
CustomLog logs/access_log common
```

The first line defines the format `common`, with the remote host specified by `%h`. The second line sends access information to a log file `logs/access_log`, using the previously defined format `common`.

Furtunately, Apache's `LogFormat` allows one to log contents of headers. By replacing the `%h` with `%{X-Real-IP}i`, the desired information is sent to the log. Therefore, normally you can simply redefine the `common` format to

```
LogFormat "%{X-Real-IP}i %l %u %t %D \"%r\" %>s %b" common
```

5.7 Debugging network traffic

Incase the traffic between client and backend must be debugged, the statement `trafficlog` *filename* can be issued. This causes the traffic to be dumped in hexadecimal format to the stated filename.

Traffic sent by the client is prefixed by a **C**, traffic sent by the back end is prefixed by a **B**. Below is a sample traffic dump of a browser trying to get a HTML page. The server replies that the page was not modified.

```
C 0000  47 45 54 20 68 74 74 70 3a 2f 2f 77 77 77 2e 63 GET http://www.c
C 0010  73 2e 68 65 6c 73 69 6e 6b 69 2e 66 69 2f 6c 69 s.helsinki.fi/li
C 0020  6e 75 78 2f 6c 69 6e 75 78 2d 6b 65 72 6e 65 6c nux/linux-kernel
C 0030  2f 32 30 30 31 2d 34 37 2f 30 34 31 37 2e 68 74 /2001-47/0417.ht
C 0040  6d 6c 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e ml HTTP/1.1..Con
C 0050  6e 65 63 74 69 6f 6e 3a 20 63 6c 6f 73 65 0d 0a nection: close..
.
. etcetera
```

```
.
B 0000  48 54 54 50 2f 31 2e 30 20 33 30 34 20 4e 6f 74 HTTP/1.0 304 Not
B 0010  20 4d 6f 64 69 66 69 65 64 0d 0a 44 61 74 65 3a  Modified..Date:
B 0020  20 54 75 65 2c 20 31 32 20 4a 75 6c 20 32 30 30  Tue, 12 Jul 200
B 0030  35 20 30 39 3a 34 39 3a 34 37 20 47 4d 54 0d 0a 5 09:49:47 GMT..
B 0040  43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 74 65 Content-Type: te
B 0050  78 74 2f 68 74 6d 6c 3b 20 63 68 61 72 73 65 74 xt/html; charset
.
. etcetera
.
```

Turning on traffic dumps will *significantly* slow down crossroads.

Besides `trafficlog`, there is also a directive `throughputlog`. This directive also takes one argument, a filename. The file is appended, and the following information is logged:

- The process ID of the crossroads image that serves the TCP connection;
- The time of the request, in seconds and microseconds since start of the run;
- A **C** when the request originated at the client, or **B** when the request originated at the back end;
- The first 100 bytes of the request.

As an example, consider the following (the lines are shortened for brevity and prefixed by line numbers for clarity):

```
1 0000594 0.000001 C GET http://public.e-tunity.com/index.html...
2 0000594 0.173713 B HTTP/1.0 200 OK..Date: Fri, 18 Nov 2005 0...
3 0000594 0.278125 B  width="100" bgcolor="#e0e0e0" valign="to...
4 0000595 0.000001 C GET http://public.e-tunity.com/css/style/...
5 0000594 0.944339 B /a></td>..  </tr>.</table>.</td><td class...
6 0000594 0.946356 B smallboxdownl">Download</td>..  <td class...
7 0000594 0.961102 B td><td class="smallboxodd" valign="top"><...
8 0000595 0.698215 B HTTP/1.0 304 Not Modified..Date: Fri, 18 ...
```

This tells us that:

- Line 1: PID 594 served a request that originated at the client. The corresponding time is (almost) 0 seconds, so this is really the start of the run.
- Line 2: A back end replied 0.17 seconds later, and 0.28 seconds later, it was still replying (this is the third line, again a **B**-type transmission).
- Line 4: PID 595 served a request that originated at the client. Again, the corresponding time is (almost) 0 seconds, since this is the first conversation part of this connection.
- Lines 5 to 7: This is the continuation of line 2. Line 7 is the last line of the **B** series (not visible from the example, but trust me, it is), so that we may conclude that it took the back end 0.96 seconds to serve the file `index.html` requested in line 1.
- Line 8: This is the answer to the client's request of line 4 (you can tell by the process ID number). So the back end took 0.68 seconds to confirm that the stylesheet requested in line 4 wasn't modified.

It is also worth while remembering that the start time of a **C** request is the time that crossroads sees the activity. Any latency between the true client and crossroads is obviously not included. This is illustrated by the below simple ASCII art:

```
client ---->---->---->--->*crossroads ====>====>====>
                                                     \
                                                  back end
```

```
                                                   /
client ----<----<----<---< crossroads ====<====<====<
```

This simple picture shows a typical HTTP request that originates at a client, travels to crossroads, and is relayed via the back end. The **C** entry in a throughput log is the time when crossroads sees the request, indicated by an asterisk. The **B** entries are the times that it takes the back end to answer, indicated by === style lines. Therefore, the true roundtrip time will be longer than the number of seconds that are logged in the throughput log: the latency between client and crossroads isn't included in that measurement.

Summarizing, the throughput times of a client-back end connection can be analyzed using the directive `throughputlog`. In a real-world analysis, you'd probably want to write up a script to analyze the output and to compute round trip times. Such scripts are not (yet) included in Crossroads.

5.8 IP filtering: Limiting Access by Client IP Address

5.8.1 General Examples

The directives `allowfrom`, `denyfrom`, `allowfile` and `denyfile` can be used to instruct Crossroads to specifically allow access by using a "whitelist" of IP addresses, or to specifically deny access by using a "blacklist". E.g., the following configuration allows access to service `webproxy` only to *localhost*:

```
service webproxy {
    port 8000;
    allowfrom 127.0.0.1;
    backend one {
        .
        . Back end definitions occur here
        .
    }
    .
    . Other back ends or other service directives
    . may occur here
    .
}
```

In this example there is a "whitelist" having only one entry: IP address 127.0.0.1, or *localhost*. (Incidentally, the same behaviour could be accomplished by stating *bindto 127.0.0.1*, in which case Crossroads would only listen to the local network device.)

In the same vein, the directive `allowfrom 127.0.0.1 192.168.1/24` would allow access to *localhost* and to all IP addresses that start with 192.168.1. The specifier `192.168.1/24` states that there are three network bytes (192, 168 and 1), and 24 bits (or 3 bytes) are relevant; so that the fourth network byte doesn't matter.

5.8.2 Using External Files

The directives `allowfile` and `denyfile` allow you to specify IP addresses in external files. The Crossroads configuration states e.g. `allowfile /tmp/allow.txt`, and the IP addresses are then in `/tmp/allow.txt`. The format of `/tmp/allow.txt` is as follows:

- The specifications follow again *p.q.r.s/mask*, where p, q, r and s are network bytes which can be left out on the right hand side when the mask allows it;
- The specifications must be separated by white space (spaces, tabs or newlines).

E.g., the following is a valid example of an external specification file:

```
127.0.0.1
192.168.1/24
10/8
```

When external files are in effect, then the signal `SIGHUP` (1) causes Crossroads to reload the external file. E.g., while Crossroads is running, you may edit `/tmp/allow.txt`, and then issue `killall -1 crossroads`. The new contents of `/tmp/allow.txt` will be reloaded.

5.8.3 Mixing Directives

Crossroads allows to mix all directives in one service description. However, some mixes are less meaningful than others. It's up to you to take this into account.

The following rules apply:

- Blacklisting and whitelisting can be used together. When combined, the blacklist will always be interpreted first. E.g., consider the following directives:

  ```
  allowfrom 192.168.1/24
  denyfrom  192.168.1.100
  ```

  Given the fact that the deny list is checked first, client 192.168.1.100 won't be able to access Crossroads. Then the allow list will be checked, stating that all clients whose IP address starts with 192.168.1 may connect. The effect will be that e.g., client 192.168.1.1 may connect, 192.168.1.2 may connect too, 192.168.1.100 will be blocked, and 10.1.1.1 will be blocked as well.

  Now consider the following directives:

  ```
  allowfrom 192.168.1.100 127.0.0.1
  denyfrom  192.168.1/24
  ```

  This will first of all deny access to all IP addresses that start with 192.168.1. So the rule that allows 192.168.1.100 won't ever be effective. The net result will be that access will be granted to 127.0.0.1.

- Blacklisting or whitelisting can be left out. A list is considered empty when no appropriate directives occur in `/etc/crossroads.conf`, or when the directive points to an empty or non-existent external file.

- Using `*from` and `*file` statements is allowed, but doesn't make sense. E.g., the following configuration sample is such a case:

  ```
  allowfrom 127.0.0.1 192.168.1/24
  allowfile /tmp/allow.txt
  ```

  There is a technical reason for this. Once Crossroads processes the `allowfile` directive, then the whole whitelist is cleared (thereby removing the entries 127.0.0.1 and 192.168.1/24), and new entries are reloaded from the file. The net result is that the `allowfrom` specification is overruled.

Crossroads only performs syntactic checking of the configuration. Some of the above samples are syntactically correct, but make no semantic sense: Crossroads doesn't warn for such situations.

5.9 Configuration examples

5.9.1 A load balancer for three webserver back ends

The following configuration example binds crossroads to port 80 of the current server, and distributes the load over three back ends. This configuration shows most of the possible settings.

```
service www {
    /* We don't need session stickyness. */
    type any;

    /* Port on which we'll listen in this service: required. */
    port 8000;

    /* What IP address should this service listen? Default is 'any'.
     * Alternatively you can state an explicit IP address, such as
     * 127.0.0.1; that would bind the service only to 'localhost'. */
    bindto any;

    /* Verbose reporting or not. Default is off. */
    verbosity on;

    /* Dispatching mode, or: How to select a back end for an incoming
     * request. Possible values:
     *   roundrobin: just the next back end in line
     *   random: like roundrobin, but at random to make things more
     *          confusing. Probably only good for testing.
     *   bysize: The backend that transferred the least nr of bytes
     *          is the next in line. As a modifier you can say e.g.
     *          bysize over 10, meaning that the 10 last connections will
     *          be used to compute the transfer size, instead of all
     *          transfers.
     *   byduration: The backend that was active for the shortest time
     *          is the next in line. As a modifier you can say e.g.
     *          byduration of 10 to compute over the last 10 connections.
     *   byconnections: The back end with the least active connections
     *          is the next ine line.
     *   byorder: The first available back end is always taken.
     */
    dispatchmode byduration over 5;

    /* Interval at which we'll check whether a temporarily unavailable
     * backend has woken up.
     */
    revivinginterval 5;

    /* TCP backlog of connections. Default is 0 (no backlog, one
     * connection may be active).
     */
    backlog 5;

    /* For status reporting: a shared memory key. Default is the same
     * as the port number, OR-ed by a magic number.
     */
    shmkey 8000;

    /* This controls when crossroads should consider a connection as
     * finished even when the TCP sockets weren't closed. This is to
     * avoid hanging connections that don't do anything. NOTE THAT when
     * crossroads cuts off a connection due to timeout exceed, this is
     * not marked as a failure, but as a success. Default is 0: no timeout.
```

```
 */
connectiontimeout 300;

/* The max number of allowed client connections. When present, connections
 * won't be accepted if the max is about to be exceeded. When
 * absent, all connections will be accepted, which might be misused
 * for a DOS attack.
 */
maxconnections 300;

/* Now let's define a couple of back ends. Number 1: */
backend www_backend_1 {
    /* The server and its port, the minimum configuration. */
    server httpserver1;
    port 9010;
    /* The 'decay' of usage data of this back end. Only relevant
     * when the whole service has 'dispatchmode bysize' or
     * 'byduration'. The number is a percentage by which the usage
     * parameter is decreased upon each connection of an other back
     * end.
     */
    decay 10;

    /* To see what's happening in /var/log/messages: */
    verbosity on;
}

/* The second one: */
backend www_backend_2 {
    /* Server and port */
    server httpserver2;
    port 9011;

    /* Verbosity of reporting when this back end is active */
    verbosity on;

    /* Decay */
    decay 10;

    /* This back end is twice as weak as the first one */
    weight 2;

    /* Event triggers for system commands upon succesful activation
     * and upon failure.
     */
    onsuccess echo 'success on backend 2' | mail root;
    onfailure echo 'failure on backend 2' | mail root;
}

/* And yet another one.. this time we will dump the traffic
 * to a trace file. Furthermore we don't want more than 10 concurrent
 * connections here. Note that there's also a total maxconnections for the
 * whole service.
 */
```

```
    backend www_backend_3 {
        server httpserver3;
        verbosity on;
        port 9000;
        verbosity on;
        decay 10;
        trafficlog /tmp/backend.3.log;
        maxconnections 10;
    }
}
```

5.9.2 An HTTP forwarder when travelling

As another example, here's my `crossroads.conf` that I use on my Unix laptop. The problem that I face is that I need many HTTP proxy configurations (at home, at customers' sites and so on) but I'm too lazy to reconfigure browsers all the time.

Here's how it used to be before crossroads:

- At home, I would surf through a squid proxy on my local machine. The browser proxy setting is then `http://localhost:3128`.
- Sometimes I start up an SSH tunnel to our offices. The tunnel has a local port 3129, and connects to a squid proxy on our e-tunity server. Hence, the browser proxy is then `http://localhost:3129`.
- At a customer's location I need the proxy `http://10.120.34.113:8080`, because they have configured it so.
- And in yet other instances, I use a HTTP diagnostic tool `Charles`[6] that sits between browser and website and shows me what's happening. I run charles on my own machine and it listens to port 8888, behaving like a proxy. The browser configuration for the proxy is then `http://localhost:8888`.

Here's how it works with a crossroads configuration:

- I have configured my browsers to use `http://localhost:8080` as the proxy. For all situations.
- I use the following crossroads configuration, and let crossroads figure out which proxy backend works, and which doesn't. Note two particularities:
  - The statement `dispatchmode byorder`. This makes sure that once crossroads determines which backend works, it will stick to it. This usage of crossroads doesn't need to balance over more than one back end.
  - The statement `bindto 127.0.0.1` makes sure that requests from other interfaces than loopback won't get serviced.

```
service HttpProxy {
    port 8080;
    bindto 127.0.0.1;
    verbosity on;
    dispatchmode byorder;
    revivinginterval 15;

    backend Charles {
        server localhost:8888;
        verbosity on;
```

---

[6] http://www.xk72.com/charles

```
        }

        backend CustomerProxy {
            server 10.120.34.113:8080;
            verbosity on;
        }

        backend SshTunnel {
            server localhost:3129;
        }

        backend LocalSquid {
            server localhost:3128;
        }
    }
```

As a final note, the commandline argument `tell` can be used to influence crossroad's own detection mechanism of back end availability detection. E.g., if in the above example the back ends `SshTunnel` and `LocalSquid` are both active, then `crossroads tell httpproxy sshtunnel down` will 'take down' the back end `SshTunnel` – and will automatically cause crossroads to switch to `LocalSquid`.

5.9.3 SSH login with enforced idle logout

The following example shows how crossroads 'throttles' SSH logins. Connections are accepted on port 22 (the normal SSH port) and forwarded to the actual SSH daemon which is running on port 2222.

Note the usage of the `connectiontimeout` directive. This makes sure that users are logged out after 10 minutes of inactivity. Note also the `maxconnections` setting, this makes sure that no more than 10 concurrent logins occur.

```
service Ssh {
    port 22;
    backlog 5;
    maxconnections 10;
    connectiontimeout 600;
    backend TrueSshDaemon {
        server localhost:2222;
    }
}
```

5.10 Linux and ip_conntrack_max

The kernel value of `ip_conntrack_max` is important for routers and balancers under Linux. Basically it's the maximum number of tracked connections. Felix A.W.O. describes the following situation:

- Crossroads seems to mark back ends as unavailable, while in fact nothing is afoot.
- This happens under heavy load.
- In `/var/log/messages` one may see the message: `kernel: ip_conntrack: table full, dropping packet.`

The reason for Crossroad's behavior is that the kernel refuses to build up a requested network connection. For Crossroads, this looks just as a non-responding back end. Crossroads therefore marks the back end as unavailable.

The solution is as follows:

- Try `cat /proc/sys/net/ipv4/ip_conntrack_max` to see the current value.
- Add something to the shown value (e.g., multiply by two), and inform the kernel of the new value, using `echo` (new-value) `> /proc/sys/net/ipv4/ip_conntrack_max`
- Make sure that the same step occurs somewhere in your boot sequence as well, or that the new valueit is stated in a configuration file such as `/etc/sysctl.conf`.

The value for *new-value* is something that you'll have to figure out yourself. Note however that each count will cause the kernel to reserve 350 bytes. So if you set `ip_conntrack_max` to 100.000, then you're already taking 33.3Mb off the total available memory.

5.11 Marking back ends as bad after more than one try

Crossroads allows you to specify on a per-back end basis how many retries are needed before a back end is considered unavailable. The default is just one, meaning that after one failed connection, Crossroads will mark a back end as unavailable (the back end may be 'revived', if you use `revivinginterval`).

Increasing the number is specified using the keyword `retries`. The following configuration defines two back ends; the one on the IP address 5.6.7.8 is somehow 'flaky', and Crossroads should try connecting 3 times before crossing it off:

```
service www {
    port 80;
    backend plugh {
        server 1.2.3.4:80;
    }
    backend xyzzy {
        server 5.6.7.8:80;
        retries 3;
    }
}
```

There may be several reasons for increasing the retries number:

- The network connections to the server may spuriously hamper, but such rare errors don't mean that the back end server is down.
- The back end server is a 'slow starter' and can't handle spikes very well. E.g., it may be a webserver which starts with only one daemon; extra capacity is added as network connections arrive, but adding capacity take a little time.

Whatever the reason, the keyword `retries` might be of help here. This keyword should however be used carefully: Crossroads will retry connecting with a small one-second delay in between. A high `retries` number means also lots of one-second delays, in which time a client is kept waiting.

6 Benchmarking

This section shows how crossroads affects the transmitting of HTML data when used as an intermediate 'station' through which all data travels.

6.1 Benchmark 1: Accessing a proxy via crossroads or directly

The benchmark was run on a system where the following was varied:

1. A website was recursively spidered through a local squid proxy. The spidering was repeated 10 times, the total was recorded.

2. Crossroads was placed in front of the squid proxy, and the website was again recursively spidered. Again, the spidering was repeated 10 times and the total was recorded.

The crossroads configuration of the second alternative is shown below:

```
service HttpProxy {
    port 8080;
    verbosity on;
    backend LocalSquid {
        server 127.0.0.1;
        port 3128;
        verbosity on;
    }
}
```

6.1.1 Results

The results of this test are that crossroads causes a negligible delay, if it is statistically relevant at all. Without crossroads, the timing results are:

```
real 0m8.146s
user 0m0.130s
sys 0m0.253s
```

When using crossroads as a middle station, the results are:

```
real 0m9.481s
user 0m0.141s
sys 0m0.230s
```

6.1.2 Discussion

The above shown results are quite favorable to crossroads. However, one should know that situations will exist where crossroads leans towards the 'worst case' scenario, causing up to 50% delay.

E.g., imagine a test where a `wget` command retrieves a HTML document from an Apache server on `localhost`. Now we have (almost) no overhead due to network throttling, hostname lookups and so on. When this test would be run either with or without crossroads in between, then theoretically, crossroads would cause a much larger delay, because it has to read from the server, and then write the same information to `wget`. Each read/write occurs twice when crossroads sits in between.

This worst case scenario will however (fortunately) occur only very seldom in the real world:

- Normally network issues, such as the above mentioned host name lookups or throughput restrictions, will add significantly to the duration of a request. The 'twice as many' read/writes caused by crossroads are then relatively irrelevant.
- Normally a significant amount of time will be spent in a back end, due to processing (e.g., when calling a servlet on a back end). Again, this processing time will weigh much heavier than the multiple read/writes.

6.2 Benchmark 2: Crossroads versus Linux Virtual Server (LVS)

LVS is a kernel-based balancer that acts like a masquerading firewall: TCP packets that arrive at the balancer are sent to one of the configured back ends. LVS has the advantage over crossroads that there is no stop-and-go in the transmission; in contrast, crossroads needs to send data via an internal buffer. Crossroads has the advantage that it offers instantaneous failover because it tries to contact the back end for upon each new TCP connection; in contrast, LVS isn't aware of downtime of back ends (unless one implements an external heartbeat). Also, crossroads offers more complex balancing than LVS.

6.2.1 Environment

On the balancer, LVS was run on port 80, its forwarding set up for two equally weighted back ends, using `ipvsadm`:

```
ipvsadm -a -t 192.168.1.250:http -r 10.1.1.100:http -m -w 1
ipvsadm -a -t 192.168.1.250:http -r 10.1.1.101:http -m -w 1
```

Crossroads was run on port 81. The configuration file is shown below:

```
service http {
    port 81;
    dispatchmode roundrobin;
    revivinginterval 5;
    backend one {
        server 10.1.1.100;
        port 80;
    }
    backend two {
        server 10.1.1.101;
        port 80;
    }
}
```

6.2.2 Tests and results

In the first test, ports 80 and 81 on the balancer were 'bombed' with 50 concurrent clients, each requesting a small page 50 times. The following timings where measured:

- How long it takes to establish a connection;
- How long it takes to retrieve the page.

The results of this test were:

- On average, each client took 0.12 seconds to connect to LVS, and each page was retrieved in 0.14 seconds;
- On average, each client took 0.11 seconds to connect to crossroads, and each page was retrieved in 0.13 seconds.

In this setup there seems to be no difference between the performance of LVS and crossroads!

In a second test, the size of the retrieved page was varied from 2.000 to 2.000.000 bytes. This test was taken to see whether crossroads would show performance degradation when transferring larger amounts of data.

For each page size, 30 concurrent clients were started, that retrieved the page 50 times. Again, the connect times and processing times where recorded.

The results of the total time (connect time + retrieval time) are shown in the below table:

Bytes	LVS timing	Crossroads timing
2000	0.130741688	0.12739582
20000	0.490916224	0.50376901
200000	3.799440328	4.33125273
2000000	45.25090855	45.9600728

Again, the results show that crossroads performs just as effectively as LVS, even with large data chunks!

7 Compiling and Installing

7.1 Prerequisites

The creation of crossroads requires:

- Standard Unix tools, such as `sed`, `awk`, `Perl` (5.00 or better);
- A POSIX-compliant C compiler;
- Support for SYSV IPC, networking and so on.

Basically a Linux or Apple MacOSX box will do nicely. To compile and install crossroads, follow these steps.

7.2 Compiling and installing

- Obtain the source distribution. It can be found on `http://crossroads.e-tunity.com`. The distribution comes as an archive `crossroads-`*type*`.tar.gz`, where *type* is `stable` or `devel`.
- Unpack the archive in a sources directory using `tar xzf crossroads-`*X.YY*`.tar.gz`. The contents spill into a subdirectory `crossroads-`*X.YY*/.
- Change-dir into the directory.
- Next, edit `etc/Makefile.def` and verify that all compilation settings are to your likings. The settings are explained in the file. **Note that** the default distribution of `Makefile.def` is suited for Linux or Apple MacOSX systems. On other Unices, or on non-Unix systems, you must particularly pay attention to `SET_PROC_TITLE_BY...`. When in doubt, comment out all `SET_PROC_TITLE...` settings. Crossroads will work nevertheless, but it won't show nice titles in `ps` listings. Also there's a macro `EXTRA_LIBS` to add linkage flags (an example for a Solaris build is included).
- Now crossroads is ready for compilation. Do a `make local` followed by `make install`. The latter step may have to be done by the user `root` if the `BINDIR` setting of `etc/Makefile.def` points to a root-owned directory.
- The documentation doesn't install in this process. If you want to install the documentation, then proceed as follows:
  - Optionally, `cp doc/crossroads.html` *htmldirectory*/; where *htmldirectory* is the destination directory for your HTML manuals;
  - Optionally, `cp doc/crossroads.pdf` *pdfdirectory*/; where *pdfdirectory* is the destination directory for your PDF manuals;
  - Optionally, `cp doc/crossroads.man` *manualdirectory*`/crossroads.1`, where *manualdirectory* is e.g. `/usr/man/man1`, `/usr/share/man1`, `/usr/local/man/man1`, `/usr/local/share/man1`. Any possibility is valid, as long as *manualdirectory* is one of the directories where manual pages are stored;
  - If your manual page system supports compressed manual pages, then you can save some space with `gzip` *manualdirectory*`/crossroads.1`.

7.3 Configuring crossroads

Now that the binary is available on your system, you need to create a suitable `/etc/crossroads.conf`. Use this manual or the output of `crossroads samplconf` to get started.

Once you have the configuration ready, start crossroads with `crossroads start`. Test the availability of your services and back ends. Monitor how crossroads is doing with:

- In one terminal, run the script:

```
while [ 1 ] ; do
    tput clear
    crossroads status
    sleep 3
done
```

Note that depending on your system you might need `sleep 3s`, i.e., with an `s` appended.

- In another terminal, run:

```
while [ 1 ] ; do
    tput clear
    ps ax | grep crossroads | grep -v grep
    sleep 3
done
```

Note that depending on your system you might need `ps -ef` instead of `ps ax`.

- In yet another terminal, run `tail -f /var/log/messages` (supply the appropriate system log file if `/var/log/messages` doesn't work for you).

Now thoroughly test the availability of your back ends through crossroads. The status display will show an updated view of which back ends are selected and how busy they are. The process list will show which crossroads daemons are running. Finally, the tailing of `/var/log/messages` shows what's going on – especially if you have `verbosity true` statements in the configuration.

7.4 A boot script

Finally, you may want to create a boot-time startup script. The exact procedure depends on the used Unix flavor.

7.4.1 SysV Style Startup

On SysV style systems, there's a startup script directory `/etc/init.d` where bootscripts for all utilities are located. You may have the `chkconfig` utility to automate the task of inserting scripts into the boot sequence, but otherwise the steps will resemble the following.

- Create a script `crossroads` in `/etc/init.d` similar to the following:

```
#!/bin/sh
/usr/local/bin/crossroads -v $@
```

The stated directory `/usr/local/bin` must correspond with the installation path. The flag `-v` causes the startup to be more 'verbose'. However, once daemonized, the verbosity is controlled by the appropriate statements in the configuration.

- Determine your 'runlevel': usually 3 when your system is running in text-mode only, or 5 when you are using a graphical interface. If your runlevel is 3, then:

```
root> cd /etc/rc.d/rc3.d
root> ln -s /etc/init.d/crossroads S99crossroads
root> ln -s /etc/init.d/crossroads K99crossroads
```

This creates startup (`S*`) and stop (`K*`) links that will be run when the system enters or leaves a given runlevel.

If your runlevel is 5, then the right `cd` command is to `/etc/rc.d/rc5.d`. Alternatively, you can create the symlinks in both runlevel directories.

7.4.2 BSD Style Startup

On BSD style systems, daemons are booted directly from `/etc/rc` and related scripts. Incase you have a file `/etc/rc.local`, edit it, and add the statement:

```
/usr/local/bin/crossroads start
```

If your BSD system lacks `/etc/rc.local`, then you may need to start Crossroads from `/etc/rc`. Your mileage may vary.