XR:
Crossroads
Load Balancer and
Fail Over Utility

Karel Kubat
2009

This document is the introductory guide, the configuration guide and the installation guide to XR. XR is an open source load balancer and fail over utility for TCP based services. It is a daemon running in user space, and features extensive configurability, polling of back ends using wake up calls, status reporting, many algorithms to select the 'right' back end for a request (and user-defined algorithms for very special cases), and much more. XR is service-independent: it is usable for any TCP service, such as HTTP(S), SSH, SMTP, database connections. In the case of HTTP balancing, XR handles multiple host balancing, and can provide session stickiness for back end processes that need sessions, but aren't session-aware of other back ends. XR furthermore features a management web interface and can be run as a stand-alone daemon, or via *inetd*.

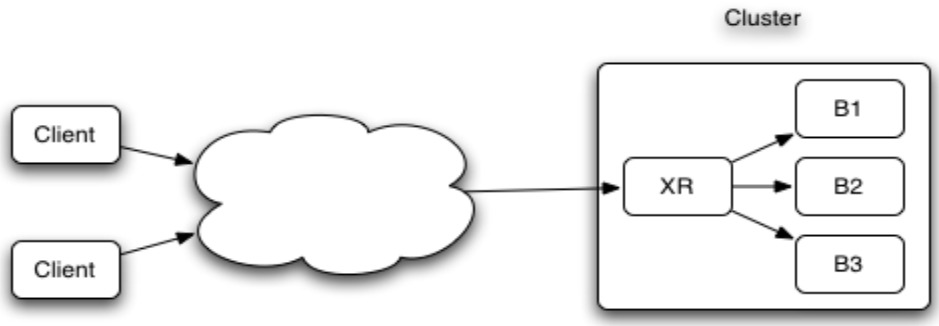
Table of Contents

1.Introduction

XR is a load balancer and fail over utility. It is typically located between network clients and a farm of servers, and dispatches client requests to servers in the farm so that processing load is distributed. Furthermore, if a server is down, next requests are routed to other servers, so that clients perceive no downtime.

Typical Usage of XR

Most often, XR is located on a separate server in a computing cluster, just in front of a farm of back ends. The below figure shows three back end servers, labeled B1 to B3. The balancer is the “entry point” as far as clients are concerned. The clients will typically access the cluster via a network, e.g. the Internet.



As far as back end servers B1, B2 and B3 are concerned, XR is their “client” who initiates contact. Towards the clients, XR acts as a “server”: it accepts their connections and makes sure that they get handled. In this mode, XR doesn't know or care what type of network data is passed; it simply shuttles bytes to and fro.

Alternatively, XR can be put into “HTTP mode”. XR then expects the “payload” to be HTTP messages. This involves more processing: the messages are unpacked and examined. In HTTP mode, XR can insert custom headers into the messages, thereby e.g. making sessions “sticky”, which causes clients to be routed to always the same back end.

Performance Benchmark

This section describes a small benchmark where “plain” throughput between a client and a web server was compared to the throughput with XR in between. The conditions were:

- An Apache web server on the local host was benchmarked. The page was 1k bytes large.
- Next, the same page was retrieved through XR running in TCP mode.
- After this, the same page was retrieved through XR running in HTTP mode.
- Finally, the HTTP balancer was put to more work by injecting X-Forwarded-For headers and stickiness cookies into the network streams.
- The same benchmark was repeated with a 10k page.

The following table shows the results.

Condition	Relative duration per page, 1k page size	Relative duration per page, 10k page size
Client connecting directly to Apache	100.00%	100.00%
Apache over XR in TCP mode	100.31%	101.23%
Apache over XR in HTTP mode	100.61%	101.53%
Apache over XR in HTTP mode, with stream modifications	100.63%	101.54%

For a number of reasons this benchmark may not be representative for other particular situations:

- For one, the client, the balancer and the the back end were located on the same system. Normally, back ends will be on different systems, and clients will be even further away – which means: longer network latencies. XR's added overhead is in those cases much smaller (with longer processing time, XR's percentage of overhead is much smaller).
- Also, in a “normal” situation, XR will not have just one back end, but will balance requests to a farm of back ends. The throughput will then of course be better than when one client hits one web server, because when XR is in the middle, multiple servers can be put to work.

However, the benchmark does show that XR is quite efficient and causes hardly any overhead.

Nomenclature

This document uses the following nomenclature:

- A **client** is any system that requests a network connection. It may be a browser, or any other program.
- A **back end** (or worker) is any system to which XR can dispatch a client request.
- A **connection** is just a TCP network connection – between a client and XR, or between XR and a back end. A **session** is a series of connections that form a 'logical' unit: XR supports sticky sessions for the HTTP protocol, which means that all connections from one client are routed to the same back end.
- XR uses a particular **dispatching algorithm** to determine which back end is best suited to handle a client connection. Dispatching algorithms include round-robin (back ends take turns), least-connections (the back end handling the least network connections is taken), and first-available (the first back end that is available is used).
- XR maintains **states** of back ends. A state tells XR whether a back end is “dead” or “alive”, and how many connections are currently active between XR and the back end. Balancing and fail-over obviously depend on the state information. XR maintains states of back end availability based on TCP-availability: if a back end accepts a network request, then it's live.
- **Wake-up calls** are periodically issued by XR to see whether unavailable back ends have come alive yet. That way, an unavailable back end can be restarted, and XR will detect this.
- A **daemon** is a process that once it starts, remains running. Daemons are typically processes that accept network connections – web servers, mail servers and the like. XR will typically act as a daemon on the balancing server.
- Most often, XR is used to handle **TCP** connections, which are just network bytes shuttled to and fro. In that case, XR doesn't know and doesn't care what the meaning is of the transmitted data (technically, this is the OSI level 5 balancing). In the special case of web

service balancing, or **HTTP,** XR can be instructed to 'peek' inside the network data and to modify it (technically, this is OSI level 7). The network data are referred to as **the payload** of the connection.

- **Stickiness** refers to the balancer's possibility to keep routing network requests from the same client to the same back end, even when the client closes its network connection and re-connects.
- **DNS** is of course the Domain Name System, which resolves host names to IP addresses. XR needs to resolve host names of the back ends it connects to, and uses caching to minimize DNS calls.

Reporting Bugs and Contact Information

XR has been extensively tested. However, it's always possible that given a particular Unix flavor, and given a specific network environment, XR shows bugs. In that case please report bugs as follows:

- Join the Crossroads forum at http://xrforum.org. The forum has a “bugs” section for reporting.
- Read the first post at the forum, it has instructions on reporting bugs.

If for whichever reason you cannot join the forum, please mail the maintainer. The address to send your mail to, appears when you run “xr -V”.

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. Crossroads is distibuted under the GNU General Public Licence, version 3. See http://crossroads.e-tunity.com for more information.

In short, the licence says that you are allowed to make modifications to the source code of crossroads, and you are allowed to (re)distribute crossroads, as long as you don't modify the licening, and as long as you also distribute 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 [KK] would appreciate hearing about them. If the changes concern new functionality or bug fixes, 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 code repository.

Technicalities

Incase you're interested in the design considerations of XR: it is a single-process multi-threaded deamon, running in user space.

The fact that XR is a multi-threaded means that once started, it hardly imposes extra requirements to the memory of the server. All actions are handled in one program image using several threads. XR however does impose requirements on CPU power: each action (client request) means processing. The processing load is not only for XR; each network request also imposes load on the kernel.

Practically however, XR performs just as well and as fast as e.g. LVS (Linux Virtual Server), a kernel-based approach that forwards TCP packets. Benchmarking shows that XR performs very well, and thanks to the fact that it's “just another user-land program”, it's extensible, scriptable and configurable.

XR is the next-generation program to Crossroads version 1.xx, which performs the same actions, but which is a forking daemon written in C (as opposed to XR which is written in C++). Also, the “bare” XR uses command line options (as opposed to a configuration file, which crossroads 1.80 used). Instead, XR has a front-end program “xrctl” to parse a configuration file, and to build up the right command line to start the balancer.

My design decisions for XR were the following:

- For improved performance, XR uses threads instead of forks to handle client request.
- Given the need for a memory-leak-free approach in a threaded program, I chose C++ above C.
- Based on my experiences with crossroads, I decided that configuration file handling, a separate parser etc., only “bloats”. All can also be specified via the command line or handled externally, and that's what XR does.
- Similarly, XR doesn't support command line options like *start* or *stop* (which crossroads requires). Such actions are easily scriptable, and hence, can be kept out of XR's code base.

Overall, XR (a.k.a. Crossroads 2) is the “lean and mean” replacement for its predecessor Crossroads 1 – optimized for speed and efficiency. I [KK] hope you like XR.

2.Obtaining and Installing XR

For the Fast and the Furious

- Get XR from http://crossroads.e-tunity.com/
- XR ships as an archive, named *xr-X.YY.tar.gz*, where X.YY is a version ID.
- Unpack the archive in a sources directory, e.g., */usr/local/src/*
- Change-dir into the created directory */usr/local/src/xr-X.YY/*
- Type *make install*, this compiles XR and installs it into */usr/sbin/*
- Fire up XR by e.g.:
  xr --verbose --server tcp:0:80 \
  --backend 10.1.1.1:80 --backend 10.1.1.2:80 --backend 10.1.1.3:80 \
  --web-interface 0:8001
  This instructs XR to listen to port 80 and to dispatch traffic to the servers 10.1.1.1, 10.1.1.2 and 10.1.1.2, port 80. A web interface for the balancer is started on port 8001.
- Direct your browser to the server running XR. You will see the pages served by one of the three back ends. The console where XR is started, will show what's going on (due to the presence of *-verbose)*.
- Direct your browser to the server running XR, but port 8001. You will see the web interface, which shows the status, and where you can alter some settings.
- Instead of starting XR by hand, consider using *xrctl*. It is installed by default to */usr/sbin*, but you must create a configuration. To do so, put the following in the file */etc/xrctl.xml:*

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system>
    <uselogger>true</uselogger>
    <logdir>/tmp</logdir>
  </system>
  <service>
    <name>web</name>
    <server>
      <address>0:80</address>
      <type>tcp</type>
      <webinterface>0:8001</webinterface>
      <verbose>yes</verbose>
    </server>
    <backend>
      <address>10.1.1.1:80</address>
    </backend>
    <backend>
      <address>10.1.1.2:80</address>
    </backend>
    <backend>
      <address>10.1.1.3:80</address>
    </backend>
  </service>
</configuration>
```

- Then type *xrctl start* to start your service, or *xrctl stop* to stop it. To monitor the activity, check the syslog output file (usually */var/log/messages*), or if you don't have the program *logger*, check */tmp/xr-web.log*. Direct your browser to port 8001 to see the web interface.

Pre-requisites

- The balancer “sec”, *xr*, runs best on a POSIX-compliant Unix system, such as Linux, MacOSX, Solaris. To compile XR, a C++ compiler and GNU Make are needed. Also, the system must be equipped with standard C libraries supporting networking, threading etc. (but that's covered in POSIX-compliancy). A Linux, MacOSX or Solaris platform equipped with *gcc*/*g*++ 4 or better will do nicely.
- The control script *xrctl* is a Perl script, so that Perl is required if you choose to use this script. (The XML parser of *xrctl* is on purpose built-in, and very simple. It suffices for the format of the configuration, and *xrctl* does not depend on Perl modules for XML parsing.)
- *Xrctl* uses a few external programs that should be present: *gzip* or *bzip2* for log file compression, and *wget* to contact the web interface. When the Perl module *LWP::UserAgent* is available, then it is used instead of *wget*.
- XR's web interface emits XML with a reference to a style sheet. This is rendered browser-side into an HTML page. All modern browsers support this. In order to modify XR's settings via the web interface, Javascript must be turned on.

3.Invoking XR

About this chapter

This chapter describes how the “bare” balancer, XR, is invoked. All command line options that drive XR are described here. This chapter is included for completeness, and for users who want to script XR invocations themselves. **Note that** there is also a control script *xrctl* which expects a configuration in XML format. If you choose to let *xrctl* control the balancer, then you don't need to be aware of XR's command line options. In that case you can just scan this chapter for concepts.

Most basic invocation

The most minimal invocation of XR is to use one option, *-b* or *--backend* to specify a back end, as in:

xr -b 10.1.1.1:80

This starts up XR to listen to the default port (10000) and to dispatch traffic to just one back end, located at the IP address 10.1.1.1, on port 80. Alternatively, one might use the form

xr --backend 10.1.1.1:80

All options in XR have a short form and a long form, e.g, *-b* and *--backend* perform identical functions. It's a matter of personal preference whether one likes the shorthand or the long form. In the remainder of this chapter, mostly the shorthand is used.

Getting Help

The command line

xr -h

shows all flags and options and can be used for a quick overview of what's possible.

Specifying Back Ends

Flag *-b* (or *--backend*) must be used at least once to specify a back end. When multiple back end specifications are used, then XR will of course distribute the load over all back ends.

Example: *xr -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80*

This defines three back ends.

Following the port specification, an optional number can be given, separated by a colon (:). When present, this is the maximum number of simultaneously allowed connections to the back end.

Example: *xr -b 10.1.1.1:80:100 -b 10.1.1.2:80:100 -b 10.1.1.3:80:100*

This defines three back ends. XR will dispatch up to 100 simultaneous connections to each back end; therefore, when all three back ends are available, XR will be allowed to service up to 300 clients. The next client will not be accepted. When only two back ends are available, XR will service up to 200 clients.

Another useful example is the following. Imagine a farm of Windows systems, to which XR dispatches Remote Desktop (RDP) connections. RDP servers on Windows can handle one user session; a new one would log out the existing session. In this case, the command:

xr -b 10.1.1.1:3389:1 -b 10.1.1.2:3389:1 -b 10.1.1.3:3389:1

would instruct XR to dispatch to three back ends, but to allow only one concurrent connection.

Specifying the Server

Towards the clients, XR acts as a server. The default is:

- A TCP server,
- Accepting connections on all interfaces (hence, on all IP addresses of the server where XR is running),
- Listening to port 10000.

Using the flag -s (--server), the server mode can be configured. Flag *-s* always has an argument with three parts, separated by a colon (:). The parts are:

- The server type: *tcp* or *http*. When the type is *http,* then XR can perform a number of extras, such as injecting headers into HTTP streams, thereby enforcing e.g. “sticky” sessions.
- The IP address to bind to. Value 0 specifies all available addresses, and e.g. value 127.0.0.1 specifies localhost-only. In that case external requests would not be serviced.
- The port specifies the listening port. The special value 0 means that XR will “listen” to the standard input stream (stdin) – which is typically used in inetd-style starting. When the port is given as 0, then the IP address to bind to is irrelevant.

Example: *xr -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80 -s tcp:0:80*. This server listens to all interfaces on port 80 and dispatches requests to three back ends. This would typically be a web server balancer.

The Dispatching Algorithm

Once a client connection is seen, XR decides which back end is best suited to handle the request. This is the dispatching algorithm. The default algorithm is: use the back end which is handling the least number of concurrent connections.

Using the flag -d the algorithm is set. The following settings are available:

- *-dl* or *-d least-connections* selects the least-connections algorithm (the default)
- *-dr* or *-d round-robin* selects round-robin dispatching, back ends take turns
- *-df* or *-d first-available* selects first-available dispatching, the first available back end is taken (in order of command line specification using flag *-b*).
- *-de:EXT* or *-d external:EXT* relies on an external program *EXT* to supply the best back end. The external program must be user-supplied, and is meant as a fallback for situations where other dispatching algorithms do not suffice. See below for a description.
- *-d h* or *-d strict-hashed-ip* takes the client's IP and “hashes” it into a preferred target back end. When the target back end is unavailable, then the connection to the client is closed.
- *-d H* or *-d lax-hashed-ip* also hashes the client's IP into a preferred target back end. However, if the back end is unavailable, then XR switches to *least-connections* dispatching.
- *-d s:SEC* or *-d strict-stored-ip:SEC* takes the client's IP and checks whether the client last connected to XR at most *SEC* seconds ago. If so, the client is dispatched to the back end that was selected during this previous visit. The back end must be available, else, the client is refused. If there was no previous visit, then the client is dispatched using the *least-connections* algorithm.

- *-d S:SEC* or *-d lax-stored-ip:SEC* takes the client's IP and checks whether the client last connected to XR at most *SEC* seconds ago. If so, the client is dispatched to the back end that was selected during this previous visit. If the back end is now unavailable, or if there was no previous visit, then the client is dispatched using the *least-connections* algorithm.
- *-d L* or *-d weighted-load* distributes requests based on XR's internal knowledge of a back end's load average. The load average must be updated “from the outside”, usually by a small job that is started from the back end. Updating XR's knowledge of a back end's average load is described in the chapter “Scripting the Web Interface”.

Stickiness by client IP

The dispatching algorithms *strict-hashed-ip, lax-hashed-ip, strict-stored-ip* and *lax-stored-ip* provide a form of “stickiness”, because clients always get dispatched to the same back end. However, when a back end becomes unavailable, stickiness will have to break – there is no way around it. Depending on the situation, one of the two approaches must be chosen:

- If it's preferable that the client is serviced at another back end, then the *lax*-... algorithm is appropriate.
- If service discontinuity is preferable to dispatching a client to a different back end than before, then the *strict*-... algorithm should be used.

There is no generic “best approach”. Whether to take the strict or the lax route, depends on on the situation.

The difference between the ...*hashed-ip* and the ...*stored-ip* algorithms is the following.

- The “hash” form inspects the IP address of a client, and converts it to a target back end. It doesn't know or care whether the back end is overloaded, or unavailable. E.g., a client IP address 192.168.1.1 might always lead to the second back end, and 192.168.1.2 to the fourth back end. The same result is achieved every time a client connects; i.e., the client with the IP address 192.168.1.1 will always go to the second back end. The disadvantage if the hash form is that it's blissfully unaware of back end conditions. But the advantage is that it's fast, and requires no storage – hence, it poses no requirements on resources.
- The “store” form is smarter. When a client connects for the first time, an appropriate back end is selected, and the obtained target is stored. The client is then dispatched to this target back end. If the client disconnects and then reconnects before a given timeout, then the same target back end is reused. The advantage is that during the connect, the “best” back end is found. The disadvantage is that this algorithm requires storage: each client's IP address is stored with the time stamp of the connection. The entry is kept in memory and is only deleted if the client hasn't connected for the specified timeout period.

Stored-ip dispatching and Reservations

The stored IP dispatch algorithms (whether strict or lax) “reserve” back ends incase clients who have disconnected come back. E.g., the dispatch mode *--dispatch-mode strict-stored-ip:600* will store the client's IP address when the client connects, and will assume that the client wants to come back to the same back end when they reconnect within 600 seconds. This is called a “reservation”.

In certain situations it may be better to wipe out existing reservations in out-of-back ends conditions. E.g., imagine 5 back ends, configured to serve a maximum of 1 client per back end. This is typical Remote Desktop Protocol (RDP) balancing. In such a case, XR may have an outstanding reservation on a back end, even when there is no actual connection to the back end. Imagine now that a new client comes along. What should XR do: honor the reservation incase a now disconnected client comes back, or dispatch the new client to the reserved back end?

By default, XR honors reservations. If you want different behavior, specify the flag *--remove-reservations* or *-G*. When this flag is present, XR will treat reservations as less important than serving new clients.

External Dispatching Algorithms

External dispatching algorithms should be only used when XR's built-ins do not suffice. When XR uses an external algorithm, then the calling of an external program will have negative impact on the performance.

When specified, XR calls the external algorithm handler using the following arguments:

- The first argument is the number of back end specifications to expect in the remainder of the command line.
- The next arguments are combo's of the back end address, its availablity, and the current number of connections. These three arguments are repeated for as many back ends as there are. Each combo consists of:
  - The back end address, e.g. *10.1.1.1:80*
  - The availability, *available* or *unavailable*
  - The number of connections to that back end

For example, given the invocation:

xr -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80 -de /where/ever/prog

the program *prog* might be invoked as follows:

/where/ever/prog 3 10.1.1.1:80 available 5 10.1.1.2:80 unavailable 0 10.1.1.2:80 available 4

This would signify that there are three back ends, of which the first one and last one are available. The available back ends have resp. 5 and 4 active connections.

The external algorithm handler must reply by printing a number on its stdout. The number is a zero-based index into the back end list (number 0 means: take the first back end in the list); therefore, the number may not exceed the number of back ends minus one. XR will try to connect to the indicated back end. E.g., if the above *prog* would reply 1, then XR would try to connect to 10.1.1.2:80 – even when this back end was initially marked unavailable. It is the external program's responsibility to make the right decision, maybe even to “wake up” a back end which XR thinks is unavailable.

As an illustration, here is a Perl script that imitates XR's “first-available” dispatch algorithm. It is meant only as an illustration: the built in algorithm is way faster. Note that the program may print 'verbose' messages on stderr, but stdout is reserved for the back end number reply.

```
# firstav.pl – sample "first available" dispatching algorithm,
# implemented as an external Perl program

#!/usr/bin/perl

use strict;

print STDERR ("firstav.pl: Invocation: @ARGV\n");
my $n = shift (@ARGV);
for my $i (0..$n – 1) {
    my $addr = shift (@ARGV);
    my $av = shift (@ARGV);
    my $conn = shift (@ARGV);
    print STDERR ("firstav.pl: $i: backend at $addr, $av, “,
            “$conn connections\n");
    if ($av eq 'available') {
```

```
        print ("$i\n");
        exit (0);
    }
}
print STDERR ("firstav.pl: Nothing available!\n");
exit (1);
```

HTTP Protocol Goodies

One of the purposes of the flag *-s* is to put XR into either *tcp* mode (the default mode) or into *http* mode. The HTTP mode is dedicated to web serving (or SOAP over HTTP etc.). In HTTP mode, XR can perform special tricks. However, HTTP mode also means that XR has to “unpack” the passing network stream, which means more overhead. As a rule of thumb, HTTP mode should be avoided if possible. XR runs “better” when it's allowed to handle HTTP as any other TCP stream. When the balancer runs in HTTP mode, then not only more processing is involved, but also, load distribution is often sub-optimal (for example, with sticky sessions, XR is forced to dispatch clients to their historical back ends, even when these back ends become overloaded).

When XR is started in HTTP mode (i.e., using flag *-s http:....*), then the following flags will instruct XR to inspect the payload and to modify it (the flags have no effect when the the server mode is *tcp*):

Sticky HTTP sessions

Flag *-S* specifies “sticky” HTTP sessions. XR will inject cookies into the HTTP stream, so that next requests from the same client are detected, and can be dispatched to the same back end (provided that the back end is still available). Technically, XR injects or inspects cookies with the name *XRTarget*.

Session stickiness is often used when back ends maintain session states of clients. If you think you need session stickiness (so that each client always gets the same back end), then maybe session data of the back ends can be shared between the back ends. PHP can save session data in a database, so that each back end has access to the same session data. Websphere can be configured to synchronize session data between nodes. When back ends share application session data, then the balancer can remain working in TCP mode. Depending on your applications, there may be other means of avoiding the necessity for HTTP balancing. Furthermore, the dispatching algorithms *strict-hashed-ip*, *lax-hashed-ip* , *strict-stored-ip* and *lax-stored-ip* may prove useful: they also provide a form of stickiness, but don't require HTTP mode.

If you are servicing devices that are not capable of cookie handling (such as some mobile platforms), then as an alternative, the back ends may return URI's with a sticky indicator, e.g. */myapp/page.html?XRTarget=0*. When XR sees such requests, and when stickiness is turned on but no sticky cookie is found, then the URI is inspected for further hints. The above URI would instruct XR to dispatch this request to the first back end (index 0). A GET parameter *XRTarget=1* would choose the second back end, and so on.

Adding X-Forwarded-For headers

Flag *-x* adds X-Forwarded-For headers to back end bound HTTP messages. The value if the header is set to the IP address of the calling client. That way the server may inspect this header value and use it for e.g., logging, or IP-based access control.

Note that XR applies header modifications only to the first block it sees on a TCP stream. Therefore, when flag *-x* is in effect and when a TCP link is used to send several HTTP requests, then only the first one will receive an X-Forwarded-For header. If you want an X-Forwarded-For header on *each* request, then downgrade the HTTP protocol to version 1.0, without keep-alives. This must be done at the back end level. For apache, the configuration that does this, is the

following:

```
SetEnv            nokeepalive
SetEnv            ssl-unclean-shutdown
SetEnv            downgrade-1.0 force-response-1.0
```

Adding XR's version ID

Flag *-X* adds a version ID header to client bound and back end bound messages. This is for debugging.

Adding custom headers

Flag *-H "Header: Value"* adds custom headers to server-directed HTTP messages. E.g., when a back end is a HTTP proxy that requires authorization, then using *-H* an authorization header can be injected.

Modifying the Host: header of the request

Flag *-I* instructs XR to override any existing Host: headers with the host name of the back end where the request is being sent to. The host name is used as it appears on the command line.

Host matching

Flag *-M HOST* states that the next back ends on the command line should only be chosen when clients request a matching host. This is one of XR's methods to serve more than one web site. The HOST argument is a regular expression. (More on this below.)

Balancing several websites

XR supports a few methods of balancing more than one "pool" of back ends on one balancer. This is referred to has multi-host balancing.

Using the IP address that XR binds to, it is possible to multi-host any type of TCP service. E.g., imagine that the balancer has two IP addresses (one of which may be virtual), 1.2.3.1 and 1.2.3.2. The address 1.2.3.1 is known in DNS as *www.onesite.org*, and 1.2.3.2 is known as *www.othersite.org*. Each site has its own back ends. It is then possible to invoke the balancer twice, to serve both sites, using something like:

```
    # Balancing of www.onesite.org
    xr --server tcp:1.2.3.1:80 --backend ... # onesite.org backends here
    # Balancing of www.othersite.org
    xr --server tcp:1.2.3.2:80 --backend ... # othersite.org backends
```

Alternatively, if the balancer server isn't equipped with several IP addresses to bind to, HTTP mode can be used in conjunction with *--host-match*. This method is of course only useful for web traffic balancing. In the HTTP mode, XR can inspect the request headers and dispatch requests based on this information. Technically, the *Host:* header of HTTP/1.0 or HTTP/1.1 requests is inspected. The flag *-M* (*--host-match*) plays a key role. The invocation may be something like:

```
xr --server http:0:80 \
   --host-match onesite \
     --backend ... \ # all back ends for "onesite" stated here
   --host-match othersite \
     --backend ... # all back ends for "othersite" stated here
```

The argument to the flag *-M* or *--host-match* is a regular expression which is matched without regard to casing. When no hosts match the request, then the client receives an error page.

Similarly, back end selection can be performed based on the URL of the request. Flag *–url-match* (*-j*) specifies this:

```
xr --server http:0:80 \
   --url-match /one \
     --backend ... \ # all back ends for url's "/one" stated here
   --url-match /two \
     --backend ...   # all back ends for url's "/two" stated here
```

Note that the argument to *--url-match* is a regular expression. The above example will assign URL's such as */what/ever/onething* to the first series of back ends (because */what/ever/onething* contains */one*). The regular expression anchor "^" can be useful here to specify a beggining-of-string. E.g., *--url-match '^/one'* would only match URL's starting with */one*.

Timeouts

Client timeouts and Back end timeouts

XR defines two timeout flags which, when in effect and exceeded, interrupt connections.

- *-t NSEC* (or *--backend-timout SEC*) is the timeout for back ends connections. When XR is trying to connect to a back end, and this value is exceeded, then the connection is terminated, and the back end is marked dead. In addition, in HTTP mode when XR is waiting for a back end's response, the answer must come within the specified time. The default is 30 seconds.
- *-T NSEC* (or *--client-timeout SEC*) is the timeout for data transfers. The default is also 30 seconds. Within this period clients must send data, or the connection is terminated.

Specific "fitting" timeouts must be used depending on the service that XR balances. E.g., the default values are well suited for web server balancing: if a client holds still for 30 seconds, then it's safe to assume that they're done.

In contrast, when XR is used to balance e.g. SSH sessions, then a much longer time out should be used, otherwise XR will interrupt the session when the end-user doesn't type for 30 seconds. In that case, *-T 7200* (2 hours) may be more appropriate. Incidentally, this method can also be used to enforce logout after a given time of inactivity. Or as yet another example, *-T 0* would be suited to balance database connections.

Read timeouts and Write timeouts

Both flags -t and -T accept either one value or two values separated by a colon (:). In the latter case the two values are taken as the "read" timeout and the "write" timeout. For example, flag *--backend-timeout 30:3* means that:

- XR will wait up to 30 seconds when the reading from the back end before deciding that the connection is dead.
- However, once data are ready to be sent to the back end, XR will only allow 3 seconds for the back end to respond and to accept written data.

Separate timeouts have the advantage that long read timeouts don't cut off inactive sessions (where nothing is being sent, so XR has nothing to read), but dead clients or back ends are detected quickly (because XR expects them to accept written information in a short period).

DNS cache timeouts

In order to minimize load on DNS servers, XR caches the resolved addresses of the back ends it connect to. The flag *-F SEC / --dns-cache-timeout SEC* controls this behavior. Each DNS entry is considered expired after SEC seconds, after which, XR will re-query the DNS to resolve the back end host name. The default timeout for DNS entries is 3600 seconds (1 hour). This may seem long, but the reasoning is that back ends are “near” to XR (in the same computing cluster) and don't change their IP address very often.

Wake-up and check-up calls

One of XR's design goals is to provide fail over: when a back end is down, clients that connect to XR are still serviced – just at an other back end. XR however needs to be able to determine when downed back ends go live again, so that they can once again be made candidates for dispatching. This is done using either wake-up or check-up calls. Wake-up calls only check back ends that XR cannot reach, and hence, believes they are down. Check-up calls are used in situations where all back ends (dead or not) should be regularly checked.

How to specify intervals

XR supports the following methods to periodically check back ends:

- *-c NSEC* (*--checkup-interval NSEC*) specifies that each NSEC seconds, XR should try verify each back end, to see if the back end is still up. If the back end accepts connections, then XR marks it as “alive”. Using this flag, simple TCP-style health checks are implemented. The default is 0, meaning that check-up calls are suppressed.
- *-w NSEC* (*--wakeup-interval NSEC*) specifies that each NSEC seconds, XR should try to verify each *unavailable* back end (i.e., a back end previously marked as dead during a check-up or during dispatching). The default is 5.

It is not possible to use both methods at the same time: this would not make sense. When checkups of all back ends are enabled, then it's not sensible to separately check the dead back ends, and v.v..

How to specify alternative back end checks

The default back end verification procedure is to connect to the back end's IP address at the back end's port. If the connection succeeds, then the back end is considered alive. However, using the flag *-g* or *--backend-check* an alternative verification can be specified.

The flag *--backend-check* must occur **before** back ends are specified using *-b* or *--backend*. The stated verification affects all next back ends on the command line (or until then next verification method is given). Alternatives for the verification are:

- *-g connect:IP:PORT*: To verify a back end, XR will connect to an alternative IP address or port. When the IP address is left out, then the back end's IP address applies. When the port is left out, then the back end's port applies.
- *-g get:IP:PORT:/URI*: To verify a back end, XR will get an HTTP document at *http://IP:PORT/URI*. The response must be an “HTTP 200 OK”. When the IP or PORT are left out, then the back end's values apply.
- *-g external:PROGRAM:* This is an escape for more complex situations. XR will invoke the PROGRAM and let it decide whether a back end is alive. The checker must print the value “0” on its *stdout* to indicate that the back end is alive. When the checker doesn't print this value, or exits with a non-zero status, the back end is assumed dead. The passed arguments are:

- The back end's address in the format IP:PORT;
- The current availability as seen by XR, either *available* or *unavailable* (unavailable back ends are either dead, or have exceeded their maximum number of allowed connections, or both);
- The current number of connections that XR serves to that back end.

The default setting is in fact *connect::*, a TCP-connect without applying alternative IP's or ports, so that each connection goes to the back end's IP address and port. As an example, consider the following invocation:

```
xr --server ..... \
   --backend-check connect::81 \
     --backend 10.1.1.1:80 --backend 10.1.1.2:80 --backend 10.1.1.3:80 \
   --backend-check get::82/healthcheck.cgi \
     --backend 10.2.1.1:80 --backend 10.2.1.2:80 --backend 10.2.1.3:80 \
   --backend-check external:/path/to/checker \
     --backend 10.3.1.1:80 --backend 10.3.1.2:80 --backend 10.3.1.3:80 \
   --backend-check get:10.10.10.10:80/verify.cgi \
     --backend 10.4.1.1 \
   --backend-check connect:: \
     --backend 10.5.1.1:80 --backend 10.5.1.2:80 --backend 10.5.1.3:80 \
   ..... # other options go here
```

The back ends would then be verified as follows:

- The back ends at 10.1.*.* would be verified by connecting to port 81. So, 10.1.1.1 would be verified by connecting to 10.1.1.1:80, 10.1.1.2 would be verified by connecting to 10.1.1.2:80 and so on.
- The back ends at 10.2.*.* would be verified by retrieving the HTTP document */healthcheck.cgi* from port 82 at the back end. So, 10.2.1.1 would be verified using *http://10.2.1.1:82/healthcheck.cgi*, back end 10.2.1.2 would be verified using *http://10.2.1.2.:82/healthcheck.cgi,* and so on.
- The back ends at 10.3.*.* would be verified using the program */path/to/checker*. An invocation might be: */path/to/checker 10.3.1.1:80 available 13*.
- The back end 10.4.1.1 would be verified using *http://10.10.10.10:80/verify.cgi*.
- The back ends at 10.5.*.* would be verified by connecting to the back end's IP and port.

An example of an external checker: MySQL

When XR balances database connections to e.g. MySQL, then an external checker is a good idea. The reason is that plain TCP connects to a MySQL port will cause the server to complain that unauthenticated connections are being made.

In this case, the following script can be used as external checker (this script is also in the distribution as *test/xr-mysql-connect*):

```
#!/bin/sh

# Where does your mysql live?
mysql=/usr/local/mysql/bin/mysql

# Get the host and port from the argument (which is the back end,
# as host:port)
host=`echo $1 | sed 's/:.*//'`
port=`echo $1 | sed 's/.*://'`

# Try to connect to the host and port. Print the result on stdout.
```

```
echo quit | $mysql -h $host -P $port --protocol=tcp --connect-timeout=3
echo $?
```

IP-based Access Control

XR supports “allow” and “deny” lists using the flags *-a* (*--allow-from*) and *-A* (*--deny-from*). Both flags take an IP address mask as argument in the “dotted-decimal” format, such as *192.168.1.255*. The address byte 255 means that the mask allows for any value in the clients' IP address in that position; e.g., 192.168.1.255 would allow 192.168.1.1, 192.168.1.2, and so on.

When access control lists for allowing and for denying are in place, then the allow-list is evaluated first. If a client matches the list, then the deny list is evaluated. If the client does not match the list, then it is serviced.

An allow list with the only value 127.0.0.1 (localhost) doesn't make much sense: instead, it might be better to bind the server to the local network device, using e.g., *--server tcp:127.0.0.1:80*.

Some combinations of allow-lists and deny-list may not make sense. E.g., the following invocation:

xr -a 192.168.255.255 -A 192.255.255.255 ....

makes no sense. First, all connections from 192.168.*.* are allowed, but then all connections from 192.*.*.* are denied. This closes the balancer to all clients.

Scripts for special actions during activity start or end

XR can run external commands (typically scripts) when a client is about to be handled at a given back end or when this finishes. These are “onstart”, “onend” and “onfail” scripts. The corresponding flags are:

- *-z CMD* or *--onstart CMD*: This will run CMD once a back end for a client is found. The CMD is passed three arguments: the client's IP address, the back end address in the form *server:port*, and the number of connections to that back end.
- *-Z CMD* or *--onend CMD*: This will run CMD once a client finishes at a back end. The arguments are the same as with the flag *-z*.
- *-y CMD* or *--onfail CMD:* This will run CMD when a client's connection to a back end aborts with a failure, e.g., due to a failure of the back end.

The onstart, onend and onfail scripts can be used in special situations, e.g., where a back end must be “prepared” just before being used. An example is a VM farm in a computer science lab, where a virtual machine is reset to a snapshot just before XR “lets through” a connection. In this example, the onstart script would be used for this purpose.

The onfail script is also run by XR when back ends are periodically checked (see *-c, --checkup-interval*). When XR fails to check up on a back end, the script is invoked with a dummy placeholder “0.0.0.0” for the client's IP address.

XR and Heavy Load

Protection Against Overloading

Flag *-m MAX* can be used to define the maximum number of simultaneous connections that XR may handle. E.g., *when -m 300* is in effect, then XR will service up to 300 concurrent clients. The next one won't be serviced.

This flag defines the maximum number of connections of XR as a whole. The back end specifier *-b...* also allows the specification of a maximum number of connections, but on a per back end level.

Hard and Soft Maximum Connection Rates

As stated above, the entire balancer can be configured to accept up to a maximum simultaneous requests. This ensures that the balancer as a whole is not brought down, and thus provides a rudimentary protection against Denial of Service attacks. However, a single client could still consume most of these connections, and "push away" other clients. In order to prevent this, XR has four flags which can limit the number of connections that originate from a single IP address.

XR uses the following terminology:

- A *soft maximum connection rate* is the number of connections in a given period, which, when exceeded, causes the offending client to be delayed. The client's connection is accepted, but not immediately serviced.
- A *hard maximum connection rate* is the number of connections in a given period, which, when exceeded, causes the offending client to be shut out. The client's connection is terminated.

Using soft- and hard max connection rates is configured using four flags.

- Flag *-U SEC* or *--time-interval SEC* sets the time interval over which maximum connections are computed. The default for *-U* is 1 (second).
- Flag *-r MAX* or *--soft-maxconnrate MAX* sets the soft max rate. E.g., when the two flags are in effect: *-U 2 -r 50,* then any client will hit its soft limit when issuing more than 50 requests per 2 seconds. When only flag *-U 20* is in effect, then any client will hit its soft limit when issuing more than 20 requests per second.
- Flag *-u USEC* or *--defer-time USEC* sets the number of microseconds by which clients will be deferred when they hit their soft limit. E.g., when *-u 500000* is in effect, then serving a client that hits their soft limit is delayed for 500.000 microseconds, which is 0.5 sec.
- Flag *-R MAX* or *--hard-maxconnrate MAX* sets the hard max rate. This rate is also computed over the interval stated with *-U*. When a client exceeds their hard maximum, then they are not served.

In addition to these flags, XR can invoke an external program or script when a given client IP exceeds the hard- or soft maxconnrate. The respective flags are *-E PROGRAM, --hard-maxconn-excess PROGRAM* and *-e PROGRAM, --soft-maxconn-excess PROGRAM*. When a rate is exceeded, XR invokes the named program with one argument: the client's IP address. The program can then e.g. invoke *iptables* to block the address.

Long running HTTP connections

When XR is balancing web servers, then under very heavy load the overall system performance will benefit when back ends downgrade HTTP connections to protocol 1.0 and close connections after each call. This increases the number of TCP connections, but each connection is much shorter than the default HTTP/1.1 way of keeping connections open. The result is that the overall count of connections per second is decreased. Downgrading HTTP to 1.0 and closing connections must be configured at the back ends. In Apache the following statements configure this:

```
SetEnv              nokeepalive
SetEnv              ssl-unclean-shutdown
SetEnv              downgrade-1.0 force-response-1.0
```

Kernel network table issues

Under very heavy load, the kernel table of network connections may become flooded with “closed” sockets that are waiting to be cleaned up – or to be re-used by the kernel.

These sockets are typically marked as *TIME_WAIT* when the command *netstat* is run. The advantage of having many sockets in this state is that the chance of re-using is higher. The disadvantage is that the entries take up space in the kernel tables; and may in fact cause errors. When the kernel table gets exhausted, then things go wrong: e.g., XR can't establish network connections to back ends, and marks them “dead” while in fact at the back end level nothing is wrong.

Sockets between XR and its back ends could therefore benefit from a longer *TIME_WAIT* period (these connections are likely to be re-used), but many short hits from clients to XR may flood the kernel tables when *TIME_WAIT* is to high. Setting the *TIME_WAIT* period to an appropriate value is therefore a matter of trial and error.

There are a few ways around avoiding kernel table overflows.

- The flag *-C* (*--close-sockets-fast*) can be used to instruct XR to set the “lingering” interval of closed sockets to 0 seconds (see the manpage for *setsockopt()* if you're interested). When this flag is in effect, then closed network sockets will be immediately cleaned up, without entering the *TIME_WAIT* state at all. The advantage is obviously that the kernel table isn't loaded with “dead” sockets. The disadvantage is that possibly re-usable kernel resources are lost.
- The operating system wide timeout can be decreased so that sockets in *TIME_WAIT* state are cleaned up faster. E.g, under Solaris, the setting for t*cp_time_wait_interval* in the file */etc/system* can be set to a shorter value than the default 4 minutes.
- The size of the sockets table of the kernel can be increased. E.g., under Linux, the following message in */var/log/messages* indicates trouble: “kernel: ip_conntrack: table full, dropping packet”. The following actions increase the value if *ip_conntrack_max*:
  - Using *cat /proc/sys/net/ipv4/netfilter/ip_conntrack_max* the current value is retrieved.
  - The kernel is informed of the new value using *echo newvalue* > */proc/sys/net/ipv4/netfilter/ip_conntrack_max*. The *newvalue* should be a number that is of course higher than the original value.

Out of Open Files

Sometimes when XR runs under high load, it will start denying clients, with a log message that there are no live back ends – while in fact, the back ends appear to run just fine. This is most often caused by an “out of open files” condition: the XR process has used up all the file descriptors that the kernel has allowed, and establishing network connections to back ends fails. XR then thinks that the back ends are “down”.

A good way of verifying whether this is the case, is to start up the web interface (see next chapter). The “activity” overview displays the number of currently opened files, and the allowed maximum.

The limit can normally be circumvented by using the “ulimit” shell command. Before starting XR, the limit for the open files is increased using:

```
ulimit –n unlimited
```

The management web interface

The flag *-W* (*--web-interface*) is used to start XR's status interface. It requires one argument, similar to the server definition: an IP address to bind to, and a port. E.g., after

xr -W 127.0.0.1:10001 -b ..... -s ......

the web interface is started on TCP port 10001, bound to the IP address 127.0.0.1, so that only local browsers can access it.

Using the web interface, it is possible to view XR's status, but also to influence key parameters of the server and the back ends. When browsing to the web interface address, then the following information is shown (the actual layout may differ and newer versions have a richer web interface):

Server 0:8000			Refresh
Type	tcp		
Dispatch mode	first-available		
Checks	Wakeup interval	sec	3
	Checkup interval	off	0
Timeouts	Client	sec	30
	Backend	sec	30
Fast sockets closing	eliminates TIME_WAIT state		no
Debugging	Verbose logging		no
	Debug logging		no
	Traffic log directory		
Network buffer size	bytes		2048
DOS Protection	Max. connections	maximum value (0 for unlimited)	0
	Time interval	sec	1
	Hard max connection rate	sessions per time interval (0 for unlimited)	0
	Soft max connection rate	sessions per time interval (0 for unlimited)	0
	Defer time	in microsec, 1.000.000 = 1 sec	500000
Access Control Lists	New allow-from		
	New deny-from		

Back end localhost:3128			
State	health	alive, available	
	connections	1	
	served	3.45783e+08 bytes, 1091 clients	
Options	weight		1
	max. connections	unlimited	0
	load average		0.1
	Up state		yes

The figure shows the overview of the server (a tcp server, listening to any IP address, on port 8000) and a back end (forwarding requests to localhost:3128, currently available, with 1 active connection, about 0.35Gb bytes served to 1091 clients). A back end which is alive can be reached by XR; a back end which is available, is alive, but also hasn't exhausted its allowed maximum connections – so it's usable.

Using the web interface, many parameters may be changed. When XR runs in HTTP mode, then HTTP-specific options appear and can also be set.

Taking a Back End Offline

When a back end needs to be taken offline, e.g., for servicing, then the web interface can be used to do this gracefully. Per back end, it displays an “up state”, which is either *up* or *down*. The default is *up*.

When the state is toggled to *down*, then XR will take the back end out of its list of candiates for dispatching. This means that the back end will not be eligible as a destination for new requests. However, existing network connections to the back will not be interrupted.

To service a back end without user impact, the following procedure can be followed:

- Using the web interface, the back end is marked as *down*.
- In the web interface, the number of connections to that back end is monitored. When this number reaches zero, there are no more running network connections to the back end.
- The back end system can then be taken offline for service.
- Once the back end is online again, the web interface is used to mark the back end as *up*. Once that happens, the back end is an eligible target for dispatching.

Adding Back Ends

Using the web interface it is possible to add back ends to a running balancer. The procedure for this is as follows:

- The back end is added using the web interface by entering its IP address and TCP port. XR will initially mark the back end as “down”.
- More options can be set, e.g., the weight or the maximum number of allowed connections.
- Finally when the back end configuration is ready, the back end is included in the dispatching cycle by toggling its state to “up”.

Deleting Back Ends

Similarly, back ends can be deleted using the web interface. It must be noted that a back end may only be deleted when its state is “down” and when there are no connections between XR and the back end. The procedure is therefore as follows:

- The back end is marked “down”.
- The web interface is monitored for the number of connections that the back end serves. This number will drop to zero, as existing connections terminate.
- When the number is zero, the “delete” button can be pressed.

Control over specific client connections

Using the web interface it is also possible to “kill” specific client connections. The web interface shows the threads that serve clients, and how long they are active, and to which back end the connection is served.

Similarly in the back end blocks, **all** running transactions to the back end can be cut off. This button should obviously be used with care.

Other Options

XR supports a plethora of other options than mentioned above. E.g., flag *-v* turns on “verbose” mode. Flag *-p* writes XR's process ID (PID) to a file, for inspection by scripts. For other options, please try *xr -h* which generates an overview. This help text shows all options and their defaults, and may be more up to date than this documentation.

4. Scripting XR: xrctl

The distribution of XR also contains a Perl script called *xrctl*. This script can be the starting point for scripting your own XR control scripts, or it can be configured and used as-is. The purpose of xrctl is to read an XML configuration file, and to build up XR-command lines to start actual balancer processes. This chapter explains how *xrctl* is used and how the configuration file */etc/xrctl.xml* is built.

Installing and configuring xrctl

The script *xrctl* is by default installed to */usr/sbin/*, so that it is immediately usable. However, no configuration file is created. You have to do that yourself. The distribution however contains the file *test/sampleconf.xml* which can be used as a starting point for your own configurations.

Configuring xrctl

The default configuration file that xrctl expects, is */etc/xrctl.xml*. Non-default configuration file locations can be selected using *xrctl -c file ...*.

The XML configuration consists of several "blocks" of information, which xrctl picks up to do its job. The blocks are described below.

General Layout

The general layout of the configuration file is shown below:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>

  <system>
    System settings so that xrctl can do its work, such as paths,
    and so on.
  </system>

  <service>
    <name>NAME</name>

    <server>
      General service descriptions, such as the IP and port that the
      balancer listens
    </server>

    <backend>
      Back end descriptions, such as the IP and port of a back end
    </backend>

    ... More backend blocks can occur here

  </service>

  ... More service blocks can occur here

</configuration>
```

Per <configuration> there can be several <service> blocks. Each <service> block will lead to the invocation of one XR balancer. Per <service> block there can be several <backend> blocks. Each <backend> block of course defines a back end for that service.

The System Block

The system block defines what xrctl needs to do its job. This block has values that are unknown and unnecessary for the balancer XR, but are relevant for xrctl only. The following settings are supported:

- *pscmd:* Optional. A “ps” command that returns lines with PID's and commands, separated by spaces. xrctl uses this to check whether services are running or not. On Linux (and MacOSX), the following is a good candidate: */bin/ps ax -o pid,command*. On Solaris, */usr/bin/ps -ef "pid comm"* can be used. When the *pscmd* is not specified, then xrctl will try to guess it.
- *uselogger:* A boolean value that states whether xrctl should try to find the “logger” program and use it for logging. When found, running balancers send their output to a pipe, connected to *logger -t SERVICENAME*, so that the information goes to syslog. When uselogger is false, or when the logger program cannot be found, then xrctl will start balancers that send output to their own log files.
- *logger:* When *uselogger* is true, then xrctl will use the stated command as the logger. The default is “logger”. You might set *uselogger* to true, and *logger* to a binary of your choice, to manage logs in the way you want. The *logger* command must accept input on stdin.
- *logdir:* When logger is not used, log files are created in this directory. The logs are named *SERVICENAME.log*.
- *prefixtimestamp*: When set to “yes” or “true”, loglines are prefixed with a millisecond-resolution timestamp. When “no” or “false”, the timestamps are suppressed. When this tag is absent, XR will try to make an educated guess, by suppressing timestamping when *logger* is used.
- *maxlogsize:* The maximum log file size before “xrctl rotate” rotates the log. This is only used when XR's logging doesn't go through “logger”.
- *loghistory:* The number of log files to keep around. Rotated logs are renamed to .log.0, .log.1 and so on. Historical logs are compressed into .bz2 or .gz when “bzip2” or “gzip” are available.
- *path:* The path along which xrctl tries to find the *xr* binary, compressors such as *bzip2* and *gzip*, and other utilities such as *wget*. If his is not specified, then xrctl will use the PATH variable from the environment.

The following listing shows an example of the <system> block:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <system>
    <pscmd>/bin/ps ax -o pid,command</pscmd>
    <uselogger>true</uselogger>
    <logdir>/var/log</logdir>
    <maxlogsize>100000</maxlogsize>
    <loghistory>10</loghistory>
    <path>/bin:/sbin:/usr/bin:/usr/sbin</path>
    <prefixtimestamp>false</prefixtimestamp>
  </system>
  ...
</configuration>
```

Service Descriptions

Each balancer instance is defined by its <service> block. The service block must contain one name definition (such as <*name*>*SERVICENAME*</*name*>), and supports sub-blocks (such as the server and back end definitions).

The name definition is stated using <*name*>*SERVICENAME*</*name*>. This identifies the service.

xrctl will start the balancer with a pseudo-command *xr-SERVICENAME* so that a “ps” listing shows easily what services are running. E.g.: <*name*>*web*</*name*> starts XR as *xr-web*. The command “*ps ax | grep xr-web*” then shows the right process.

The balancer server description is contained in a <*server*> block. This block must at least state the balancer type (tcp or http), the IP and port that the balancer listens to:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      <address>0:20010</address>
      <type>http</type>
       ... More server descriptions
    </server>
  ... More service descriptions
  </service>
</configuration>
```

The dispatch mode is stated in a <*dispatchmode*> tag. The default is *least-connections* (see the description of the XR's command line flag *--dispatch-mode*). For example:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      <address>0:20010</address>
      <type>http</type>
      <dispatchmode>lax-hashed-ip</dispatchmode>
    </server>
  </service>
</configuration>
```

How to treat reservations is defined in the tag <*removereservations*>. This is only relevant in stored-ip dispatching. The default is not to remove reservations, but to honor them:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <dispatchmode>strict-stored-ip</dispatchmode>
      <removereservations>true</removereservations>
    </server>
  </service>
</configuration>
```

The back end, client, and DNS cache timeouts can be specified using their tags *clienttimeout*, *backendtimeout* and *dnscachetimeout*. In the following example, DNS entries are valid for 2 hours, read/write timeouts for clients are 20 seconds, and read/write timeouts for back ends are 10 seconds:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      <address>0:20010</address>
      <type>http</type>
```

```
      <clienttimeout>20</clienttimeout>
      <backendtimeout>10</backendtimeout>
      <dnscachetimeout>7200</dnscachetimeout>
    </server>
  </service>
</configuration>
```

The flags *clientreadtimeout*, *clientwritetimeout*, *backendreadtimeout* and *backendwritetimout* can be used to specify separate timeouts for reads and writes:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      <address>0:20010</address>
      <type>http</type>
      <clientreadtimeout>20</clientreadtimeout>
      <clientwritetimout>5</clientwritetimeout>
      <backendreadtimeout>10</backendreadtimeout>
      <backendwritetimeout>1</backendwritetimeout>
    </server>
  </service>
</configuration>
```

Wakeup and checkup intervals are defined in a block *checks*. When absent, the wakeup interval is 5 (seconds), and there are no checkups (value 0). Example:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <checks>
        <wakeupinterval>3</wakeupinterval>
        <checkupinterval>0</checkupinterval>
      </checks>
    </server>
  </service>
</configuration>
```

Onstart, onend and onfail commands can be defined in <onstart> and <onend>:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <onstart>/where/ever/onstart-script</onstart>
      <onend>/where/ever/onend-script</onend>
      <onfail>/where/ever/onfail-script</onfail>
    </server>
  </service>
</configuration>
```

When necessary, TCP buffer sizes are controlled using <*buffersize*> (set to 4k in the below example):

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
```

```
  <service>
    <name>webone</name>
    <server>
      ...
      <buffersize>4096</buffersize>
    </server>
  </service>
</configuration>
```

The block <*debugging*> can optionally increase XR's verbosity, turn on debug messages, and turn on traffic logging. The default is all “off”.

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <debugging>
        <verbose>yes</verbose>
        <debug>yes</debug>
        <logtrafficdir>/tmp</logtrafficdir>
      </debugging>
    </server>
  </service>
</configuration>
```

The tag <*closesocketsfast*> can be used to immediately discard TCP sockets, instead of letting them slip into a TIME_WAIT state. This can be turned on when the balancer's OS runs out of free sockets:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
   ...
      <closesocketsfast>yes</closesocketsfast>
    </server>
  </service>
</configuration>
```

Access control lists (allow-from and deny-from masks) are stated in a block <*acl*>. When absent, XR will allow all client IP's and deny none. For example, the following configuration allows all clients from the IP's 10.*.*.* and from 192.168.*.*, but denies one IP 192.168.1.100:

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <acl>
        <allowfrom>10.255.255.255</allowfrom>
        <allowfrom>192.168.1.255</allowfrom>
        <denyfrom>192.168.1.100</denyfrom>
      </acl>
    </server>
  </service>
</configuration>
```

Basic denial of service (DOS) protection is stated in a block <*dosprotection*>. Connections from clients are counted over intervals of *timeinterval* seconds. When the number exceeds the *hardmaxconnrate* then the clients' connections are closed. When the number exceeds the *softmaxconnrate* then the clients are serviced, but first they are delayed for *defertime* microseconds. The overall number of connections that the balancer may accept, is defined by *maxconnections*. For example, the following configuration states that:

- The balancer may accept up to 400 connections.
- Connecting clients are traced during intervals of 2 seconds.
- When a client exceeds 200 attempts in the 2 seconds, then their connection is closed.
- When a client exceeds 150 connections in the 2 seconds, then they are delayed for 1 second before being serviced.

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <dosprotection>
        <timeinterval>2</timeinterval>
        <hardmaxconnrate>200</hardmaxconnrate>
        <softmaxconnrate>150</softmaxconnrate>
        <defertime>1000000</defertime>
        <maxconnections>400</maxconnections>
      </dosprotection>
    </server>
  </service>
</configuration>
```

Additionally, tags *softmaxconnexcess* and *hardmaxconnexcess* can be used to instruct XR to invoke an external program when a client exceeds a limit. E.g., consider the below configuration. When a client, say 10.1.1.1, connected more than 200 times per 2 seconds, then XR would (a) drop the connection, (b) invoke */path/to/program 10.1.1.1*.

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
      ...
      <dosprotection>
        <timeinterval>2</timeinterval>
        <hardmaxconnrate>200</hardmaxconnrate>
        <maxconnections>400</maxconnections>
        <hardmaxconnexcess>/path/to/program</hardmaxconnexcess>
      </dosprotection>
    </server>
  </service>
</configuration>
```

HTTP-specific directives are placed in a block <*http*>. These directives are only active when the server type is *http*. For example, the following configuration:

- Does not inject a header *"XR: version-id"*; (where the version-id is XR's version, such as 2.23). This is also the default;
- Injects back end bound headers *"X-Forwarded-For: client-ip"*;
- Does not use sticky HTTP sessions, this is also the default;

- Modifies the Host: header of all requests to the server name of the current back end;
- Injects two specific server headers.

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
    <type>http</type>
    ...
      <http>
        <addxrversion>off</addxrversion>
        <addxforwardedfor>on</addxforwardedfor>
        <stickyhttp>off</stickyhttp>
        <replacehostheader>on</replacehostheader>
        <serverheaders>
          <header>MyFirstHeader: Whatever</header>
          <header>MySecondHeader: WhateverElse</header>
        </serverheaders>
      </http>
    </server>
  </service>
</configuration>
```

Back end definitions

Following the <*server*> block, back ends are described in their <*backend*> blocks. Each back end definition can state:

- The back end address, in the format <*address*>*IP:PORT*</*address*>;
- An optional weight, in the format <*weight*>*NUMBER*</*weight*>. The default weight is 1. The bigger the number, the bigger the server; e.g., back ends with weight 2 are assumed to be able to handle twice the load of back ends with weight 1, before XR will consider them equally loaded.
- A maximum number of connections, in the format <*maxconnections*>*NUMBER*</*maxconnections*>. The default is 0, which means: unlimited.
- A back end checking mode, e.g. <*backendcheck*>*get:IP:PORT/URI*</*backendcheck*>
- In the case of HTTP balancing (i.e., when the server type is “http”), a <*hostmatch*> tag can be stated. When present, its value is taken as a regular expression. Only requests for hosts that match the regular expression are dispatched to the back end.
- Similar to host matching, the tag <*urlmatch*> defines which back ends are candidates for which URL's. The value is a regular expression.

For example, the following two <*backend*> blocks define two back ends. The first back end is a large server (twice as big as the second one), is allowed to handle up to 300 connections, and may serve requests for *www.mysite.org*. The second server is allowed to handle up to 100 connections, and may serve requests for *www.myothersite.org*. The first back end (at server1:80) is checked using TCP-connects to the back end's address itself. The second back end is checked by retrieving http://server2/healthcheck.cgi. This must return HTTP/1.x 200 OK if the back end is to be considered “live”.

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
    <type>http</type>
    ...
```

```
    </server>
    <backend>
      <address>server1:80</address>
      <weight>2</weight>
      <maxconnections>300</maxconnections>
      <hostmatch>www.mysite.org</hostmatch>
    </backend>
    <backend>
      <address>server2:80</address>
      <maxconnections>100</maxconnections>
      <backendcheck>get:server2:80/healthcheck.cgi</backendcheck>
      <hostmatch>www.myothersite.org</hostmatch>
    </backend>
  </service>
</configuration>
```

URL matching is shown in the following example. Back end *server1* is the back end of choice for all URL's except */img*. All URL's */img* are handled by the second back end *server2*.

```
<?xml version="1.0" encoding="UTF-8">
<configuration>
  <service>
    <name>webone</name>
    <server>
    <type>http</type>
    ...
    </server>
    <backend>
      <address>server1:80</address>
      <urlmatch>^/[^i][^m][^g]</urlmatch>
    </backend>
    <backend>
      <address>server2:80</address>
      <address>server1:80</address>
      <urlmatch>^/img</urlmatch>
    </backend>
  </service>
</configuration>
```

Using xrctl

Once *xrctl* is configured, services are started using *xrctl start*. Optionally, only a subset of all configured services can be started. E.g., given the above configuration, *xrctl start webone* would only start the service identified *by* <*name*>*webone*</*name*>, even when *more* <*service*> blocks occur in the configuration. The same applies for all other actions of *xrctl*: when an extra argument is present, it denotes a specific service; while no argument means: all services.

The actions of *xrctl* are:

- *xrctl configtest*: Builds an XR command line from the XML configuration and tests it;
- *xrctl list*: Lists which services are configured;
- *xrctl start*: Starts services;
- *xrctl stop*: Stops services. The services may linger until the running connections terminate;
- *xrctl kill*: Kills services: stops them, also interrupting any running connections;
- *xrctl force*: Forces services “up”: not yet started services are started;
- *xrctl stopstart*: Restarts services (useful for e.g. configuration changes);
- *xrctl killstart*: Kills and stops services;

- *xrctl status*: Shows which services are running and which not;
- *xrctl rotate*: Rotates log files (when *xrctl* logs to separate log files and not via *logger*).
- *xrctl generateconfig:* Queries the web interfaces of the running services and shows the configuration derived from actual run time values.

When *xrctl* has started the balancer(s), then the following applies:

- XR processes are started, with process names that reflect the service. E.g., given the above example a process *xr-webone* is started. (This is actually the binary *xr,* but the distinct name is easier to find in the process list.)
- The process ID of the web balancer can be found in */var/run/xr-webone.pid,* because the <*piddir*> is set to */var/run*.
- When *xrctl* ends output to *logger,* then all verbose messages, errors or reports are collected in */var/log/messages* (or another syslog-file, depending on the Unix variant, e.g., */var/log/system.log* under MacOSX).
- When *xrctl* sends output to private log files, then messages, errors and reports are collected in */var/log/xr-webone.log,* because the <*logdir*> says so.
- To generate a back end report to the log file, the following actions are necessary:
  - The service name must be known – and hence, the process name. E.g., for service *web* the process name is *xr-web*.
  - This process is sent a SIGHUP signal, using *killall -1 xr-webone*. Alternatively the process ID can be looked up in */var/run/xr-webone.pid*, and *kill -1 process-id* can be used.
- To view the report online, a browser can be directed to *http://BALANCER-IP:20001/*. This is the address of the web interface (because <*webinterface*> says so).

5. Running XR

The basic operation of XR is that it listens to a file descriptor (either a network socket or stdin), until activity is detected. Once a request from a client is seen, it is dispatched to a back end.

Errors and reporting messages are always reported to *stderr*, prefixed by ERROR or REPORT. When flag *-v* is in effect, then messages about new connections etc. are also sent to *stderr*, prefixed by INFO. For debugging there is also a flag *-D*. When in effect, debugging messages are sent to stderr, prefixed by DEBUG.

XR does not daemonize by itself – i.e., it doesn't "go into the back ground". It is the job of the invoker to make sure that this happens (if it's requested). Such actions are easily scriptable.

Demo Mode

Given three webservers at 10.1.1.1, 10.1.1.2 and 10.1.1.3, the following command can be used to balance traffic on port 80 in a "demo mode":

xr -v s tcp:0:80 -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80

Because flag *-v* is in effect, the terminal where XR is started will show messages about XR's activity. XR can then be shutdown by pressing ^C.

Daemon Mode, invoked from the command line

Using the bare XR invocation

The standard program *logger* is a very useful tool that catches the output from an other program, and sends it to a syslog-defined log file. To fire up XR in daemon mode, a command such as the following one does the trick:

xr -v -s tcp:0:80 -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80 2>&1 | logger -t xr-web &

The parts of the invocation are:

- *xr -v -s tcp:0:80 -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80* is the basic XR start up command,
- The flag *-v* instructs XR to show what's going on (verbose mode),
- *2>&1* makes sure that stderr and stdout are combined;
- *| logger* makes sure that XR's combined output goes to *logger*,
- *-t xr-web* is an argument to *logger* which causes syslog-messages to be prefixed by "xr-web", for easier viewing in the log file,
- The final ampersand daemonizes the entire command.

If *logger* is unavailable, then the following command does the trick as well:

xr -s tcp:0:80 -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80 2>&1 >> /var/log/xr-web.log &

This simply directs all input to */var/log/xr-web.log*.

Using xrctl

Crossroads can also be started using the control script *xrctl*. The command is:

xrctl start

This of course requires that the configuration */etc/xrctl.xml* is present.

Resilience

The balancer program XR hasn't been known to “crash”, but anything is possible. The control script *xrctl* has a specific command “force” that starts services unless they are already running. The command can be invoked via *cron* to periodically check that XR is running, and if not, to start it. The *crontab* configuration is as follows (see *man crontab* for a description of the format):

```
# Periodically check that XR is running
* * * * * /usr/sbin/xrctl force >/dev/null
```

This checks that XR is running every minute. If the one minute period is too long, then the following simple trick can be used to check e.g. twice per minute:

```
# Periodically check that XR is running
* * * * * /usr/sbin/xrctl force >/dev/null
* * * * * sleep 30; /usr/sbin/xrctl force >/dev/null
```

Inetd Mode

To start XR via inetd, the invocation only changes in that the listen port is set to 0. In that case, XR will assume that clients connect via stdin and stdout.

The procedure is as follows:

- In the file */etc/services* a service is added to link XR with a given TCP port, say 80:
  xr 80/tcp # Crossroads load balancer for web services
- A small intermediate script, say */usr/sbin/xr.sh* is created, with the command line that startx XR:

  ```
  #!/bin/sh
  exec /usr/sbin/xr -s tcp:0:0 -b 10.1.1.1:80 -b 10.1.1.2:80 -b 10.1.1.3:80
  ```

  Note the usage of option *-s tcp:0:0* where the “port number” 0 instructs XR to serve requests on stdin. Having the intermediate script *xr.sh* is by no means necessary, but it does make it easier to add back ends or other options to XR by simply editing the script.
- Alternatively, if *xrctl* is used, the <*address*> in the server description is set to <*address*>*0:0*</*address*>. The intermediate starter */usr/sbin/xr.sh* is then:

  ```
  #!/bin/sh
  exec /usr/sbin/xrctl start
  ```

- In the file */etc/inetd.conf* the invocation is specified for service *xr*:
  xr stream tcp nowait root /usr/sbin/xr.sh xr.sh
  This wil start XR as user *root* when *inetd* sees activity on port 10000. Note that instead of *root* any other user is permitted.
- The program *inetd* is restarted (using e.g. *killall -1 inetd*).

Xinetd Mode

Xinetd is an inetd-replacement and uses separate configuration files for each service.

- In the file */etc/services* a service is added to link XR with a given TCP port, say 80:
  xr 80/tcp # Crossroads load balancer for web services
- A small intermediate script, say */usr/sbin/xr.sh* is created, with the command line that startx XR (see above).

- In the directory */etc/xinetd.conf* a new file is created named *xr* and having the following contents:

```
service xr
{
 socket_type = stream
 protocol = tcp
 wait = no
 user = root
 server = /usr/sbin/xr.sh
}
```

  Instead of *root* another user can be specified.

- The program *xinetd* is restarted using *killall -1 xinetd*

Interpreting the Verbose Output

When XR is started with the flag *-v,* then informational messages are printed to stderr. Depending on the invoking command line, these messages will go to syslog, a private log file, the console, or similar.

XR always prefixes the messages by a thread ID and by the word INFO, DEBUG, WARNING or ERROR. The thread ID is shown to distinguish messages from each other when several threads are running. E.g., here's an example of informational messages that are generated when the “verbose” option is set.

```
0xa00fafa0 INFO: Invoking command line: xr/build/xr --backend server1:22
--backend server2:22 --backend server3:22 --verbose --server tcp:0:2222
0xa00fafa0 INFO: XR running as PID 90112
0xa00fafa0 INFO: TCP server for balancer listening to 0.0.0.0:2222
0xa00fafa0 INFO: Initial backend state: server1:22 is available
0xa00fafa0 INFO: Initial backend state: server2:22 is available
0xa00fafa0 INFO: Initial backend state: server3:22 is available
0xa00fafa0 INFO: Starting wakeup thread.
0xa00fafa0 INFO: Awaiting activity on fd 4
0xa00fafa0 INFO: Accepted connection from 127.0.0.1 as client fd 5
0xa00fafa0 INFO: Current back end states:
0xa00fafa0 INFO: Back end server1:22: 0 connections, status available
0xa00fafa0 INFO: Balancer is serving 0 clients
0xa00fafa0 INFO: Back end server2:22: 0 connections, status available
0xa00fafa0 INFO: Balancer is serving 0 clients
0xa00fafa0 INFO: Back end server3:22: 0 connections, status available
0xa00fafa0 INFO: Balancer is serving 0 clients
0xb0103000 INFO: Dispatch request for client fd 5
0xb0103000 INFO: Dispatching client to back end 0 server1:22
0xb0103000 INFO: Dispatching client fd 5 to server1:22, fd 5
0xa00fafa0 INFO: Accepted connection from 127.0.0.1 as client fd 8
0xa00fafa0 INFO: Current back end states:
0xa00fafa0 INFO: Back end server1:22: 1 connections, status available
0xa00fafa0 INFO: Balancer is serving 1 clients
0xa00fafa0 INFO: Back end server2:22: 0 connections, status available
0xa00fafa0 INFO: Balancer is serving 1 clients
0xa00fafa0 INFO: Back end server3:22: 0 connections, status available
0xa00fafa0 INFO: Balancer is serving 1 clients
0xb0185000 INFO: Dispatch request for client fd 8
0xb0185000 INFO: Dispatching client to back end 1 server2:22
0xb0185000 INFO: Dispatching client fd 8 to server2:22, fd 5
0xb0185000 INFO: Done dispatching client fd 8 at server2:22
```

Most above messages are prefixed with 0xa00fafa. This is the “main thread”, which also shows the invocation command line. At some point a client request is picked up as file descriptor 5,

and handled by thread 0xb013000. Later on, a second client request is picked up as file descriptor 8, and handled by thread 0xb01185000.

The numbers are irrelevant, except that they are used to distinguish the separate threads.

Signals: Reporting and Stopping

The following signals are interpreted by XR:

Signal 1 (SIGHUP) causes XR to report its status on stderr. Depending on the invocation this will go to syslog, the console, or a separate log file.

Below is a sample of such a report, generated using *killall -1 xr*:

```
0xa00fafa0 REPORT: *** XR STATUS REPORT STARTS ***
0xa00fafa0 REPORT: Back end server1:22:
0xa00fafa0 REPORT:   Status: available, alive
0xa00fafa0 REPORT:   Connections: 1 (max 0)
0xa00fafa0 REPORT:   Served: 15938 bytes, 2 clients
0xa00fafa0 REPORT: Back end server2:22:
0xa00fafa0 REPORT:   Status: available, alive
0xa00fafa0 REPORT:   Connections: 1 (max 0)
0xa00fafa0 REPORT:   Served: 6529 bytes, 1 clients
0xa00fafa0 REPORT: *** XR STATUS REPORT ENDS ***
```

This report shows that XR knows two back ends, at server1:22 and on server2:22. The report shows that both back ends are available and alive:

- **Alive** means that the back end can be reached; i.e., that it responds to network requests. In the above example both reports are alive.
- **Available** means that the back end is alive, isn't flagged as “down”, and that a restriction of a maximum number of connections has not been reached. In the above example both back ends have no connections limit (indicated by “max 0”), and hence they are also available whenever they are alive and not flagged down.

The report furthermore shows how many bytes each back end has handled, and how many clients were served. (Note that reports can also be generated using the web interface, see the description of option *-W* or *--web-interface*.)

All other signals request the termination of XR's balancing. XR will stop accepting connections, and will wait until all clients have been served, and will then exit.

Restarting XR

This section describes how to restart the core balancer program XR while minimizing downtime.

Restarting using the bare XR command line

The section is included for completeness. Most often, the XR binary itself won't be started on the command line, but *xrctl* will be used.

In non-production environments, it may suffice to stop XR and to type in a command line to start it again using other options (e.g., with more back ends). In production environments however, stopping the balancer means not accepting client requests until a new XR is started – which in turn means a short black out. For obvious reasons this isn't desirable.

XR doesn't allow restarting in the sense that the main server is kept alive, while it re-reads its configuration (there are technical reasons for this fact). However, avoiding black outs is easily achieved using one of the following approaches:

1. If you need to modify an option, try the web interface first. Modifications via the web interface cause no downtime at all; new requests are served using the new values. In fact most, if not all runtime options can be modified through the web interface.
2. If an option can be modified via the web interface, then it can also be modified using a scripted URL call, e.g., using wget. For example, "*wget http://localhost:8001/server/maxconnections/300*" will modify the maximum number of allowed connections to the balancer (assuming that there's a web interface active at http://localhost:8001). See the chapter "Scripting the web interface" for more information.
3. If you are using *xrctl* to control XR, you can edit the configuration file and supply new values, and then type *xrctl stopstart*. This will gracefully let the balancer finish its work and will then start the balancer using new values. This is explained in detail below.
4. Finally, you can use the following command line approach. When XR is stopped, then underway requests are served, while the listening port is freed up – so that immediately an other XR can be started.

   E.g., imagine that an XR instance is running, invoked by the following command line and dispatching to two back ends:

   ```
   xr --server http:0:80 --backend 10.1.1.1:80 --backend 10.1.1.2:80
   ```

   Now the server type should be changed to "tcp" (actually in more recent versions of XR, this can be done using the web interface – but this example is included here for illustration purposes). This can be achieved using the command line:

   ```
   killall xr; \
   xr --server tcp:0:80 --backend 10.1.1.1:80 --backend 10.1.1.2:80
   ```

   Following the *killall* command, the previous balancer is instructed to continue serving existing connections. A new balancer is immediately started to serve new connections.

   The flag *-n* is intended to test XR invocations for syntactic correctness, while not actually starting XR. This flag is typically used in scripts (such as *xrctl*). Using this safeguard, the best approach would be the following:

   - The new command line is tested, using:
     ```
     xr -n --server tcp:0:80 \
           --backend 10.1.1.1:80 --backend 10.1.1.2:80
     ```
   - When XR reports that the command line is correct, the new invocation is made active:
     ```
     killall xr; \
     xr --server tcp:0:80 \
        --backend 10.1.1.1:80 --backend 10.1.1.2:80
     ```

Restarting using xrctl

There are basically two ways of modifying configurations of running services: (a) The configuration file */etc/xrctl.xml* is modified, and XR is restarted; (b) The running XR services are modified using the web interface, and a new configuration is generated for posteriority.

When the configuration file of an XR service must be changed, then *xrctl* can be used as follows. As an example we assume that a service named *web* needs to changed from type *tcp* to http (note again that this can also be done using the web interface, but the "restarting" method is described nevertheless for illustration).

- Using *xrclt status web* it is verified that the service is running. *Xrctl* should reply with:

Service web: running.

- The configuration file */etc/xrctl.xml* is edited, and the type is changed to:

```
<type>http</type>
```

- The service is restarted, using *xrctl restart web*.

Changes that can be applied via the web interface don't require restarts and hence, there is zero possibility of downtime. As an example we assume that a service needs to be extended with a new back end. The procedure is then as follows:

- Using the web interface, a new back end is created. The back end is fully configured and then switched to "on".
- In order to save the configuration for the next time, the following commands are executed:

```
xrctl generateconfig > /tmp/newconfig.xml && \
    xrclt –c /tmp/newconfig.xml configtest && \
    mv /tmp/newconfig.xml /etc/xrctl.xml
```

The shell command separator "&&" ensures that each next command is run only if the previous command has succeeded. Hence, the following actions are performed:

1. The configuration is re-generated from all running balancers (as stated in */etc/xrctl.xml*). This new configuration is saved in */tmp/newconfig.xml*. If this fails, then the sequence stops here.
2. If the generation of the new configuration has succeeded, then it it verified using "configtest". Note the flag *-c /tmp/newconfig.xml* which instructs *xrctl* to operate on that specific file. If this fails, then the sequence stops here.
3. If the new configuration verified successfully, then */etc/xrctl.xml* is replaced by the new configuration.

For extra verification, *xrctl list* can be used to review the suggested XR commands. For example, for a visual inspection of the way that a new configuration would run the balancer XR, consider the commands:

```
xrctl generateconfig > /tmp/newconfig.xml
xrctl list
xrctl list –c /tmp/newconfig.xml
```

This shows how XR would be started given the old configuration, and given the new one. The difference between the shown command lines must account for all changes that have been applied to XR using the web interface.

6. Scripting the web interface

The web interface can of course be used for automated monitoring or adjusting XR. It actually serves XML in a fairly simple format which is comparable to the XML of the configurations for *xrctl*. The web interface accepts GET requests to standardized URI's to modify variables.

The overview which is presented in XML format can be seen using e.g. "*wget -O- http://localhost:10001*" (provided that the web interface runs on port 10001). An abbreviated example is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="/xslt"?>
<status>
  <server>
    <address>0:8000</address>
    <type>tcp</type>
    <maxconnections>0</maxconnections>
  </server>
  <backend>
    <nr>0</nr>
    <address>localhost:3128</address>
    <weight>1</weight>
    <maxconnections>0</maxconnections>
    <live>alive</live>
    <available>available</available>
    <connections>3</connections>
    <bytesserved>68527</bytesserved>
    <clientsserved>3</clientsserved>
  </backend>
  <activity>
    <thread>
      <id>0xb0289000</id>
      <description>Serving</description>
      <backend>0</backend>
      <address>localhost:3128</address>
      <duration>1.12</duration>
      <clientip>1.2.3.4</clientip>
    </thread>
  </activity>
</status>
```

The web interface accepts calls to special URL's (other than /) to modify key parameters. After modification, the web interface returns the same status overview in XML format. The special URL's are shown in the below table. The following placeholders are used in the URL descriptions:

- NUMBER is of course a number.
- BOOL is a boolean value. Use "yes, true, on" or "no, false, off", or a number. A number "0" is taken as "no".
- INDEX is a non-negative integer which is used to point out a back end, or a header, or another list-element. The value of the index is zero-based and can be seen in the XML status output. E.g., given the above XML snippet, the back end at localhost:3128 has index 0.

URL	Description
Overview	
/	Returns status overview in XML format.

URL	Description
/xslt	Returns the built-in transformation sheet to render the XML output to HTML,
General server controls	
/server/type/tcp	Sets the balancer server type to “tcp”.
/server/type/http	Sets the balancer server type to “http”.
/server/maxconnections/NUMBER	Sets the server-wide maximum connections. Use 0 for unlimited.
/server/clienttimeout/NUMBER	Sets the number of seconds after which a client's network connection is considered dropped. Use 0 to suppress timing out.
/server/backendtimout/NUMBER	Sets the number of seconds after which a back end's connection is considered timed out.
/server/dnscachetimeout/NUMBER	Sets the number of seconds after which a DNS lookup of a back end hostname is considered expired. Set 0 to suppress back end hostname caching.
/server/wakeupinterval/NUMBER	Sets the interval in seconds for wakeup checks. Use 0 to turn off.
/server/checkupinterval/NUMBER	Sets the interval in seconds for checkups. Use 0 to turned off. When turned on, wakeups are turned off automatically.
/server/closesocketsfast/BOOL	Controls whether TCP sockets are closed without entering the TIME_WAIT state.
Onstart, onend and onfail scripts	
/server/onstart/SCRIPT	XR will run the SCRIPT upon activity start. The script arguments are a client's IP address and a back end (server:port).
/server/onstart/	Removes the onstart definition.
/server/onend/SCRIPT	XR will run the SCRIPT upon activity end. The arguments are as with onstart.
/server/onend/	Removes the onend definition.
/server/onfail/SCRIPT	XR will run the SCRIPT when periodic checks of a back end fail, or when the handling of a client fails. The IP address argument is 0.0.0.0 during periodic check failure, or the true client IP adress during client handling.
/server/onfail/	Removes the onfail definition.
HTTP Goodies controls (applicable when the server type is “http”)	
/server/addxforwardedfor/BOOL	Controls the injection of Add-X-Forwarded-For headers in HTTP mode.
/server/stickyhttp/BOOL	Controls sticky sessions in HTTP mode.
/server/replacehostheader/BOOL	Controls whether the Host: header should be replaced with the back end server name.

URL	Description
/server/newheader/HEADER	Defines a new header for injection in HTTP mode. Use %20 to code a space in the HEADER value, as in *X-MyHeader:%20MyValue*
/server/changeheader/INDEX	Deletes header INDEX from the list of injected headers in HTTP mode.
/server/changeheader/INDEX/HEADER	Changes header INDEX. The new HEADER value will be injected in HTTP mode.
Verbosity and Debugging	
/server/verbose/BOOL	Controls verbose message generation.
/server/debug/BOOL	Controls debugging message generation.
/server/logtrafficdir/	Stops traffic logging.
/server/logtrafficdir/DIRECTORY	Starts traffic logging to DIRECTORY. Encode slashes in the directory name as %2F, as in *%2ftmp%2flogdir*.
DOS Protection	
/server/timeinterval/NUMBER	Sets the time interval over which connections rates are computed to NUMBER seconds.
/server/hardmaxconnrate/NUMBER	Sets the “hard” max connection rate to NUMBER sessions per time interval.
/server/softmaxconnrate/NUMBER	Sets the “soft” max connection rate to NUMBER seconds per time interval.
/server/defertime/NUMBER	IP's that exceed the soft max connection rate will be deferred (stalled) for NUMBER microseconds before being served.
/server/hardmaxconnexcess/PROGRAM	States the program to invoke upon excess of the hard maxconn rate. Program is invoked with the client's IP address as argument.
/server/hardmaxconnexcess/	Removes the setting.
/server/softmaxconnexcess/PROGRAM	States the program to invoke upon excess of the soft maxconn rate.
/server/softmaxconnexcess/	Removes the setting.
Access Control Lists	
/server/addallowfrom/MASK	Adds MASK to the allow-from list. The MASK is e.g. 10.11.255.255, which would allow the IP's from the B-class network 10.11.*.*.
/server/allowfrom/INDEX/	Removes the index'th element of the allow-from list.
/server/allowfrom/INDEX/MASK	Changes the MASK of the index'th element of the allow-from list.
/server/adddenyfrom/MASK	Adds MASK to the deny-from list.
/server/denyfrom/INDEX/	Removes the index'th element of the deny-from list.

URL	Description
/server/denyfrom/INDEX/MASK	Changes the MASK of the index'th element of the deny-from list.
Adding or deleting back ends	
/server/addbackend/IP:PORT	Adds a “vanilla” back end at IP:PORT to the back end list. The back end is initially marked “down”.
/server/deletebackend/INDEX	Deletes index'th back end from the list. The back end must be marked “down” and may have no active connections.
Back end specific controls	
/backend/INDEX/weight/WEIGHT	Sets the weight of the indexed back end.
/backend/INDEX/maxconnections/NUMBER	Sets the maximum connections for the back end. Use 0 for NUMBER to indicate “unlimited”.
/backend/INDEX/hostmatch/EXPRESSION	Sets the host matching expression for the given back end. Only meaningful in HTTP mode. Use '.' for the expression to allow the back end to be used for all host requests.
/backend/INDEX/urlmatch/EXPRESSION	Sets the URL matching expression for the given back end. Only meaningful in HTTP mode. Use '.' for the expression to allow the back end to be used for all URL requests.
/backend/INDEX/loadavg/FLOAT	Updates XR's internal knowledge of a back ends load average. Used with dispatch mode “weighted-load”.
/backend/INDEX/up/BOOL	Updates the up/down state of the back end. When down, the back end is excluded from dispatching.
/backend/INDEX/backendcheck/VALUE	Changes the back end checking. VALUE has the same format as flag *--backend-check:* e.g., *connect:IP:PORT, get:IP:PORT/URI, external:PROGRAM*.
/backend/INDEX/stopconnections	Unconditionally stops all connections to the indicated back end. Use with care, as this cuts off clients.
Single client / thread related	
/thread/kill/THREADID	Stops the indicated thread. The thread id may be either decimal, or a hexadecimal value preceded by 0x. Use with care, as this cuts off a client.

Scripting example: Requiring ping-able clients

The following problem was brought to me by Rajeev S. (thanks for reporting). Sometimes, clients will “die”, their network connection will disconnect, without a proper network shutdown. If the back end doesn't close the connection after a given time of inactivity, then a situation may arise where XR doesn't detect a “dead” client, and keeps the connection open. For example, imagine that:

- XR is used to balance RDP connections;
- A client connects through their firewall which is configured to silently drops packets (“stealth mode”);
- This clients suddenly dies.

The RDP server will periodically send a “are-you-there” signal to see if the client is still alive. However, since the client's firewall silently discards network data, there will be no answer. The RDP server will assume that the client is alive, and will not close the connection. Hence, so will XR, resulting in a “dead” connection.

Such situations can be perfectly scripted. The approach is the following:

- The web interface is queried to get an activity overview.
- The activity shows thread ID's and connected client IP addresses.
- Now, the client is pinged. If it doesn't respond, then the web interface is used to kill the connection.

This approach is coded in a small Perl script which is in the distribution as *test/xr-client-ping*. The script is started with two arguments: the web interface address, and a number of seconds which states the checking interval. The script happily queries the web interface, tries to ping clients, and if this fails, tells the web interface to stop the connection.

Note that this approach in no way puts extra stress on XR. The overhead on XR of periodically querying the web interface is negligible. All other extra actions that this test requires (running Perl, pinging) occur outside of XR.