# XR:
# Crossroads
# Load Balancer and
# Fail Over Utility

Karel Kubat
2008

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 can provide session stickiness for back end processes that need sessions, but aren't session-aware of other back ends. XR can be run as a stand-alone daemon, or via *inetd*.

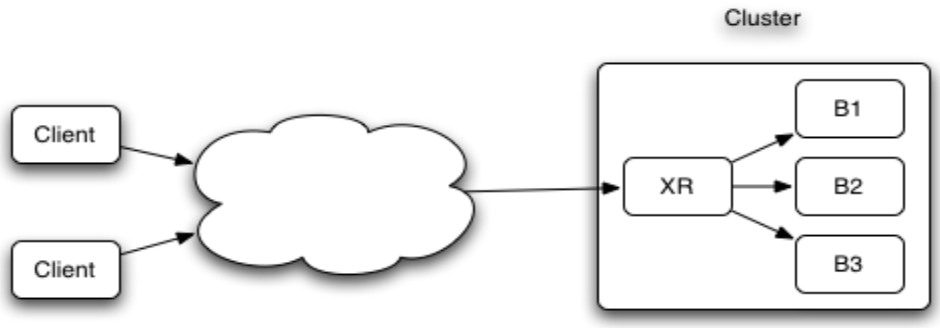
# Table of Contents

# 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

The following table show the results of a short benchmark of the following conditions:

- An Apache web server on the local host was benchmarked, with XR running as an idle process.
- 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.

| Condition | Average time (sec) | Percentage |
|---|---|---|
| Plain Apache | 0.00388 | 100% |
| Apache over XR in TCP mode | 0.00404 | +4.1% |
| Apache over XR in HTTP mode | 0.00449 | +15.7% |
| Apache over XR in HTTP mode, with stream modifications | 0.00462 | +19.1% |

This benchmark is for a number of reasons the worst case for XR usage. 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). Another reason is that this test involved repetitively retrieving a static, very small HTML page (30 bytes), resulting in many short TCP bursts. Normally, network connections will shuttle many more bytes, and will require more server processing – so that, again, XR's overhead will be relatively smaller.

## 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.

## 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:

- Determine your version of XR and the e-mail address of the maintainer using *xr -V*.
- Mail the maintainer with a detailed bug report. Include XR's version number, verbose output of XR (collected with flags *--verbose* and *--debug)*, and include the command line that invoked XR.

e-tunity hosts a mailing list, dedicated to XR, to which you can subscribe. Questions about configurations, options, how-to subjects, etc., can be directed at the list. Please see http://crossroads.e-tunity.com for the list address and for instructions how to join.

## 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

XR 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. Linux LVS (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 1.80, which performs the same actions, but which is a forking daemon written in C (as opposed to XR which is written in C++). Also, XR uses command line options (as opposed to a configuration file, which crossroads 1.80 used). 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, 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 is the "lean and mean" replacement for its predecessor crossroads – optimized for speed and efficiency. I [KK] hope you like XR.

# 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*
  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.
- 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)*.
- For a status report, issue *killall -1 xr*. The console where XR was started, will show which back ends are active, how many bytes were transferred, etc..
- Instead of starting XR by hand, copy the control-script *xrctl* to a directory of your choice, e.g. */usr/sbin*. Edit *xrctl* and configure your service(s) at the top. Then type *xrctl start* to start all your services, or *xrctl stop* to stop them.

## Pre-requisites

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.

# Invoking XR

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 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*, 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 inject 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.

Using the IP address that XR binds to, it is also possible to multi-host web server balancing. 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 configure the balancer 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 here
```

## 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.

## 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 connects 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.

## 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 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.

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:

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

- Flag *-H* 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 inserted.

The flags have no effect when the server type isn't *http*.

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 sub-optimal (as XR is forced to dispatch clients to their historical back ends, even when these back ends become overloaded).

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* or *lax-hashed-ip* may prove useful: they also provide a form of stickiness, but don't require HTTP mode.

## Timeouts

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

- *-t NSEC* 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* 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.

Summarizing, the timeout flags *-t* and *-T* influence XR as follows:

- Flag *-T* is always the longest "silence" period of clients. When clients do not send data during the specified time, then the connection is terminated.
- Flag *-t* is always the maximum connection time to back ends. When a back end don't accept requests from XR within the specified time, then the connection it terminated and the back end is marked dead.
- In HTTP mode (i.e., when flag *--server http*:.... is in effect), flag *-t* is also the longest "silence" period for server responses. When XR waits for a response from the back end, and this period is exceeded, then XR interrupts the connection. The server is however not marked dead; XR just assumes that it took too long to process the request.

## Wake-up and check-up calls

XR supports the following methods to periodically check back ends:

- *-c NSEC* specifies that each NSEC seconds, XR should try to connect to 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* specifies that each NSEC seconds, XR should try to connect to each unavailable back end (i.e., a back end previously marked as dead during a check-up or during dispatching). The default is 5.

Theoretically both flags can be used simultaneously; e.g., it's possible to define check-up calls each minute, with a wake-up call interval of 5 seconds.

## 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.

## 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.

## XR and Heavy Load

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/ip_conntrack_max* the current value is retrieved.
  - The kernel is informed of the new value using *cat newvalue > /proc/sys/net/ipv4/ip_conntrack_max*. The *newvalue* should be a number that is of course higher than the original value.

## 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.

# 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

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*.

## 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:
  */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.
- 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:
  */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.
- 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. The thread ID is shown to distinguish messages from each other when several threads are running. E.g., here's an example:

```
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 localhost:22:
0xa00fafa0 REPORT:   Status: available, alive
0xa00fafa0 REPORT:   Connections: 1 (max 0)
0xa00fafa0 REPORT:   Served: 15938 bytes, 2 clients
0xa00fafa0 REPORT: Back end 127.0.0.1:22:
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 localhost:22 and on 127.0.0.1:22. 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, 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.

The report furthermore shows how many bytes each back end has handled, and how many clients were served.

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

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 the fact that when XR is stopped, 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 a third back end at 10.1.1.3:80 must be added, while avoiding a black out. This can be achieved using the command line:

*killall xr; *
*xr --server http:0:80 --backend 10.1.1.1:80 --backend 10.1.1.2:80 --backend 10.1.1.3:80*

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

# 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.

## Installing xrctl

The script *xrctl* is by default **not** installed by the make process. The script must be manually installed by, e.g., copying it to */usr/sbin/*.

## Configuring xrctl

Before using *xrctl*, the script must be opened using an editor and the configuration options (at the top of the script) must be set. Relevant options are:

- *$piddir* is a directory where process ID stamp files are stored. Actions such as *xrctl stop* read the process ID's. The default is */var/run*.
- *$pscmd* is a "ps" command that outputs process ID's and commands. When *xrctl* fails to find PID files, it reads the output of this command to find out under which process ID's XR is running. The default is */bin/ps ax -o pid,command*. This setting works for Linux and MacOSX.
- *$use_logger* indicates whether *xrctl* should use *logger* to send XR's output to syslog. The default is 1. When *$use_logger* is zero, or when the program *logger* cannot be found, then *xrctl* redirects XR's output to separate logs.
- *$logdir* is the directory where log files are stored. This value is only used when *$use_logger* is 0, or when *logger* is unavailable.
- *$maxlogsize* is the maximum log file size. When *xrctl rotate* is run, log files bigger than this value are "rotated".
- *$log_history* is the number of history logs to keep around. When rotating, log files are renamed to *file.0*, the previous *file.0* to *file.1* and so on. The last kept file is indicated by *$log_history*.
- *$bindirs* is an array of directories where executable programs are searched. *xrctl* thus tries to find XR itself and the programs "logger", "bzip2", "gzip".
- *%services* is the definition of network services to balance. The details of this setting are described below.

The hash *%services* defines how many, and which network services XR will balance. A separate XR instance is invoked for each service, with the right flags *--server, --backend* and so on. The structure is best shown by example. The following definition defines two entries, arbitrarily named *web* and *ssh*. Per entry all relevant XR flags are given. Per flag zero, one or more values are given (flags without options, such as *--verbose* have zero values).

```
my %services =
  (
   'web' =>
   { '--server'              => [ qw(http:0:80) ],
     '--backend'             => [ qw(10.0.0.1:80 10.0.0.2:80 10.0.0.3:80) ],
     '--verbose'             => undef,
     '--add-x-forwarded-for'=> undef,
   },
   'ssh' =>
   { '--server'              => [ qw(tcp:0:22) ],
     '--backend'             => [ qw(server1:22 server2:22 server3:22) ],
     '--verbose'             => undef,
```

```
    '--client-timeout'      => [ qw(7200) ],
  },
);
```

The per-service options can be specified using the long flags (e.g. *--server*) or using the short form (e.g. *-s*).

## 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 ssh* would only start the SSH balancer, not the web balancer. The same applies for most 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 list*: Lists which services are configured;
- *xrctl start*: Starts services;
- *xrctl stop*: Stops services;
- *xrctl force*: Forces services "up": not yet started services are started;
- *xrctl restart*: Restarts services (useful for e.g. configuration changes);
- *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*).

When *xrctl* has started, then the following applies:

- XR processes are started, with process names that reflects the service. E.g., given the above example two processes *xr-web* and *xr-ssh* are started. (Both are actually the binary *xr,* but the distinct name is easier to find in the process list.)
- The process ID's of the service balancers can be found in */var/run/xr-web.pid* and */var/run/xr-ssh.pid* (unless *$piddir* is configured as another directory than */var/run*).
- When *xrctl* ends output to *logger,* then all verbose messages, errors or reports are collected in */var/log/messages* (or another syslog-file, depending on the Unix variant, e.g., */var/log/system.log* under MacOSX).
- When *xrctl* sends output to private log files, then messages, errors and reports are collected in */var/log/xr-web.log* and */var/log/xr-ssh.log* (unless *$logdir* is configured as another directory than */var/log*).
- To generate a back end report, the following actions are necessary:
  - The service name must be known – and hence, the process name. E.g., for service *ssh* the process name is *xr-ssh*.
  - This process is sent a SIGHUP signal, using *kilall -1 xr-ssh*. Alternatively the process ID can be looked up in */var/run/xr-ssh.pid*, and *kill -1 process-id* can be used.
  - The report is looked up in either the syslog file (e.g., */var/log/messages*) or in *xrctl*'s own log */var/log/xr-ssh.log*.
  - Usually it's more practical to issue *tail -f /var/log/messages* or *tail -f /var/log/xr-ssh.log*, and then to issue *killall -1 xr-ssh*.

## Configuration Changes

When the configuration of an XR service must be changed, then *xrctl* can be used as follows. As an example we assume that the above shown service *web* must be extended with a new back end 10.0.0.4:80.

1. Using *xrclt status web* it is verified that the service is running. *Xrctl* should reply with: *Service web: running*.
2. The script *xrctl* is edited, and the configuration of the service is modified. The relevant section with an added back end should contain the line:
   '*--backend*' => *[ qw(10.0.0.1:80 10.0.0.2:80 10.0.0.3:80 10.0.0.4:80) ],*
3. The service is restarted, using *xrctl restart web*.