YXORP

Guide and Reference Manual

version: stable-1.23.2

November 12, 2006

Introduction

A reverse proxy, as seen from the outside, acts like a normal server would. Unlike a server, it does however not respond to requests directly; instead, it forwards the requests it receives to one or more servers. A client does not directly communicate to a server; all traffic must pass trough the reverse proxy. The reverse proxy can, before deciding to forward the request to a server, perform all kinds of checking on the request contents, and optionally modify the request, or even reject it.

Yxorp is a reverse proxy for HTTP. To a client, like a web browser, Yxorp behaves like a server; to a server it behaves like a client. The main focus in the design and development of Yxorp is to check the validity of the HTTP traffic. Yxorp does this by taking apart each request, examining each field before the request is reassembled and sent on towards the web server. The reverse also applies: a response from the server is also broken down, examined and reassembled. Both requests and responses can be modified by Yxorp under control of rules. These rules are made up of settings and statements in a special programming language, and allow to check and modify all fields.

Many reverse proxies are available, but most are derived from normal (forward) proxies, or modules included in a web server. Yxorp, by contrast, has been a reverse proxy from the start, with no intention to ever be a forward proxy or web server. Also, many commonly used reverse proxies are commercial products; one of the reasons Yxorp was started was a comment from a friend 'that there were no good open source reverse proxies'.

Table of Contents

Introduction....2
How to deploy YXORP....6
  Minimal network setup....6
  Secure network setup....6
  YXORP box setup....8
  Domain names....8
  SSL certificates....9
Basic configuration tutorial....10
  Basic setup....10
    Testing the configuration....12
  Adding SSL....12
    Testing the configuration....12
  Adding basic authentication....13
    Testing the configuration....13
  Adding support for non-standard headers....14
    Testing the configuration....14
  Adding load balancing....14
    Testing the configuration....15
  Adding a sorry rule....16
    Testing the configuration....16
  Sticky load balancing....16
    Testing the configuration....17
  Sticky loss rules....17
    Testing the configuration....18
Rule progamming examples....20
  Enforcing SSL for certain requests....20
  SSL to backend servers....20
  Filtering on client IP address, range, or domain name....21
    Filtering on a single IP address....21
    Filtering on an IP range....21
    Filtering on domain names....22
  Using black-, white- and greylisting....22
    Blacklisting....22
    Whitelisting....23
    Greylisting....24
  Using an application to authenticate access to other URI's....25
    Situation....25
    Example....25
    Conclusion....27
Configuration reference....28
  Overview of the configuration file....28
  Syntax of the configuration file....28
  Configuration file contents....29
    Configuring listeners....29
      Reconfiguring....30
      Example....30
    Configuring servers....31
      Reconfiguring....31
      Example....31
    Configuring virtual servers....32
      Reconfiguring....32
      Example....32
    Configuring real servers....33
      Reconfiguring....33

Example....33
Configuring rules....34
Reconfiguring....34
Example 34
Configuring logging....34
Reconfiguring....36
Example....36
Configuring a configuration listener 36
Reconfiguring....37
Example....37
Configuring threads....37
Reconfiguring....37
Example....37
Configuring security....37
Reconfiguring....38
Example....38
Configuring basic authentication....38
Reconfiguring....38
Example....38
Configuring global settings....38
The header table....40
Reconfiguring....41
Example....41
Rule language reference....43
Introduction....43
Statements....43
Variables....43
Expressions and operator precedence....44
Special variables....45
Rule syntax....45
Program....45
Block....45
Statements 46
If....46
Foreach....46
Whitelist....46
Blacklist....47
Return....47
Expression....47
Identifier....48
Numbers....48
Literals....48
Indirect variable....48
Regular expressions....49
Function reference 49
basic_auth_check(realm, “local”) – enforce basic authentication....49
clientinrange(range) – check if client IP is in a range....49
clientinip6range – check if client IP is in an IPv6 range....50
clientstate(type) – retrieve value from client state....50
concat(...) – concatenate 50
contains(haystack, needle) – test if needle contains haystack....50
contains_characters(string, list) – test if string contains characters in list....50
containscase(haystack, needle) – test if needle contains haystack....51
enumerate_dupvar(variablename) – enumerate duplicate variables....51
enumerate_reqhdr() – enumerate request header variables....51
enumerate_rsphdr() – enumerate response header variables....51
equal(s, r) – compare two variables....51

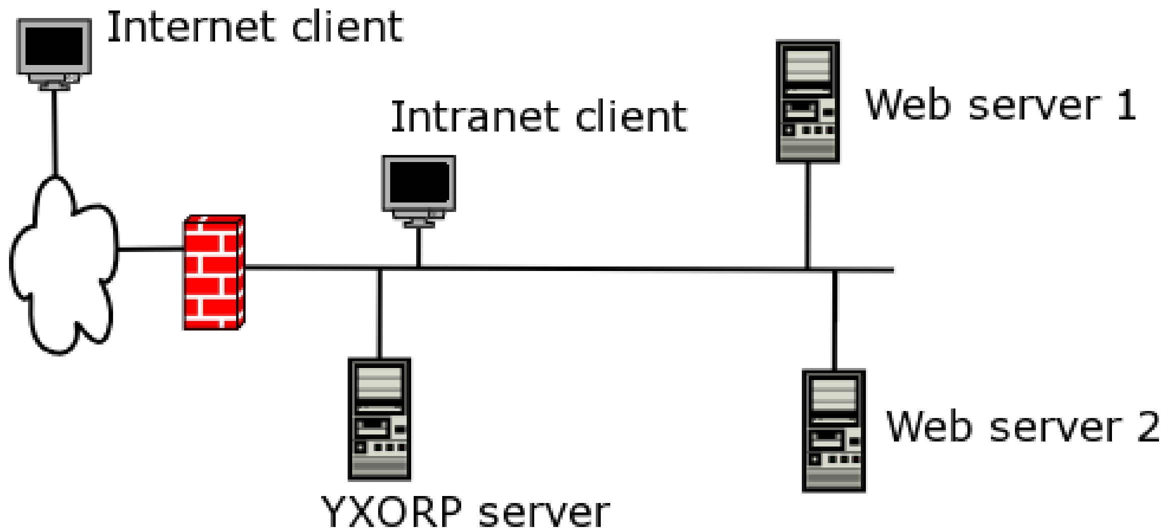
equalcase(s, r) – case insensitive compare of two variables....51
findheader(header) – retrieve contents of non-standard headers....51
getattr(variablename, attr) – test attributes of a variable....51
getclientip() – get the client ip address....51
getclientdomainname() – get the client domain name....51
getclientstatedvar(variablename) – read a variable from the client state....52
getlength(variablename) – returns the length of a variable....52
getlistenername() – returns the listener name....52
getmaxcount(variablename) – return the number of variables in a dupvar group....52
getmaxlength(variablename) – return the maxlength attribute of a variable....52
getorder(variablename) – return the order attribute of a variable....52
getoriginalname(variablename) – return the originalname attribute of a variable....52
getserverturnaround() – return the turnaround time for the server request....52
getsslbackend() – return the sslbackend flag....52
getsticky() – return the sticky flag....52
isssIsession() – return the SSL state of the client session....52
redirect(url) – redirect request....52
reject(reason) – reject request....52
setattr(variablename, type) – set variable attribute....53
setclientstatedvar(variablename, value) – set a variable in the client state....53
setcookiedomain(string) – set cookie domain....53
setdup(variablename, value) – set duplicate variable....53
setmaxlength(variablename, length) – set variable max length attribute....53
setorder(variablename, ordinal) – set variable order attribute....53
setoriginalname(variablename, originalname) – set variable originalname attribute....53
setsslbackend() – set ssl backend....53
setsticky() – set sticky flag....53
statuscodemsg(code) – returns status code textual message....54
trace(message) – send a message to the trace log....54
unsetsslbackend() – clear ssl backend....54
unsetsticky() – clear sticky flag....54
yesno(x) – return truth value as yes or no....54
The virtual machine....54

How to deploy YXORP

Yxorp, as a reverse proxy, should be somewhere between the client browsers and the web server(s) it is proxying for; the traffic between the clients and the web server(s) should flow through Yxorp. To make this work, and especially if you want to use Yxorp as a security device, you will need to carefully plan for the network environment that you will be using Yxorp in. If you make the wrong decisions, implementing Yxorp may decrease security instead of enhancing it; but also, if the networking design is wrong, it may cause your web servers to become unreachable, or serve pages with some of the content missing etc.

Minimal network setup

The diagram below shows a typical example of the minimal Yxorp deployment. The diagram shows an Ethernet segment which connects web servers, an Yxorp server, and client systems. A firewall connects to the Internet; there are also clients on the Internet.



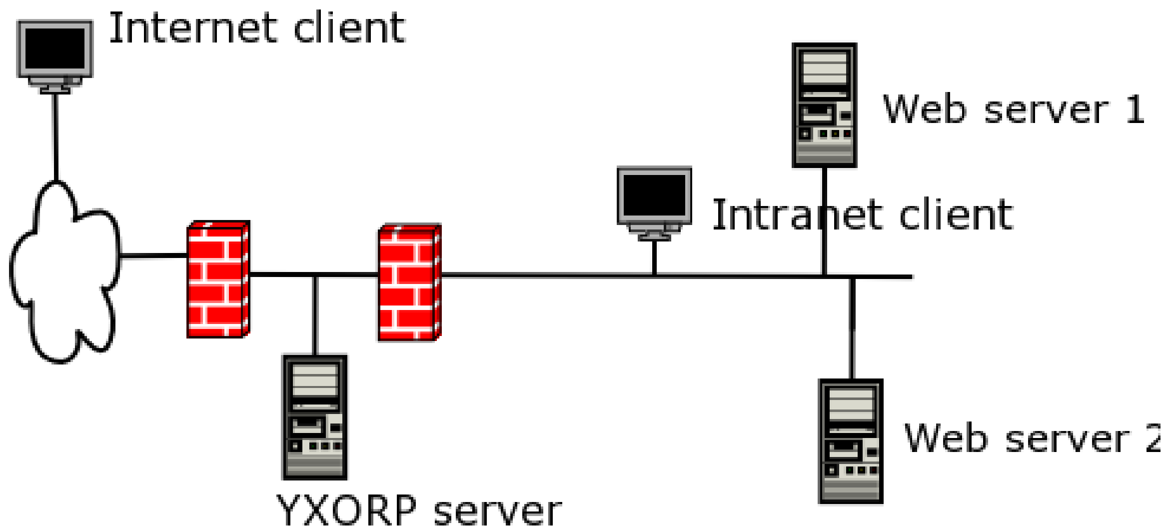
The firewall is, in this kind of setup, most probably running NAT (and thus will probably have only one public Internet address).

An important note to take from this picture is that this setup is not inherently secure. The firewall would need to be configured to allow traffic into the Yxorp server (i.e. port 80 and/or 443). If Yxorp is broken, this allows traffic into your network without further restrictions. If, for instance, Yxorp is compromised and the attacker is able to insert another configuration, it would be possible to setup a tunnel that might work well enough to allow other TCP protocols to be passed into any box on your network.

A possible solution to this is to make sure all boxes on your network are secure in themselves, i.e. at least they run firewalling software and are adequately patched.

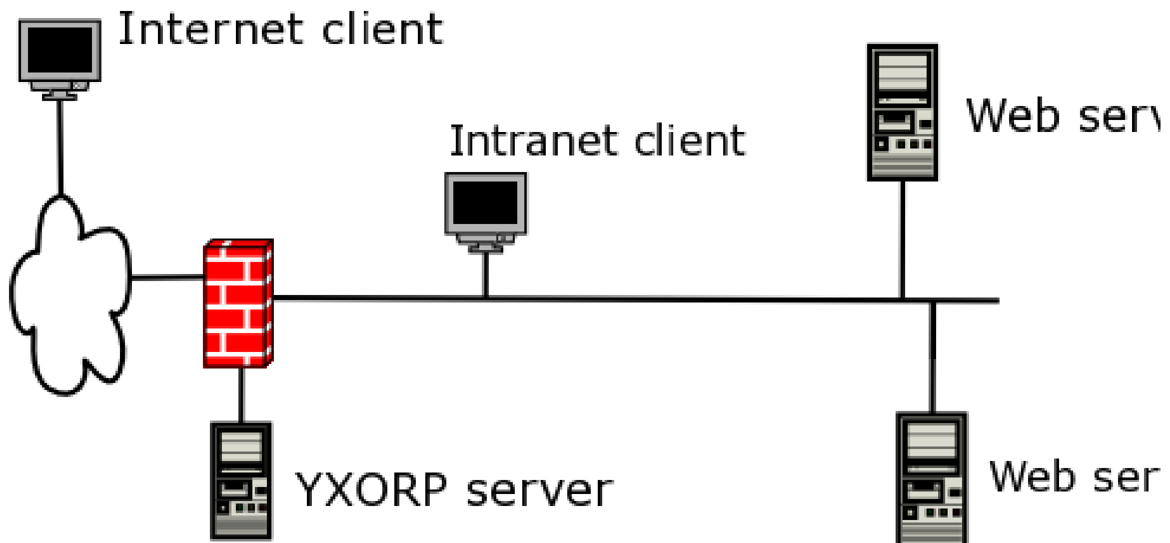
Secure network setup

The diagram below shows a more secure setup. The Yxorp box is sitting between two firewalls; the outer firewall allows port 80 and/or 443 from the Internet into Yxorp; the inner firewall allows port 80 and/or 443 from Yxorp into the web servers.



The difference with the minimal setup is that the traffic between Yxorp and the web servers is now restricted by the second firewall; this means that even if Yxorp is compromised, it is still not possible to use that vulnerability to reach the Intranet clients or other systems on the internal network (like your server running Samba and NFS for your MP3-collection).

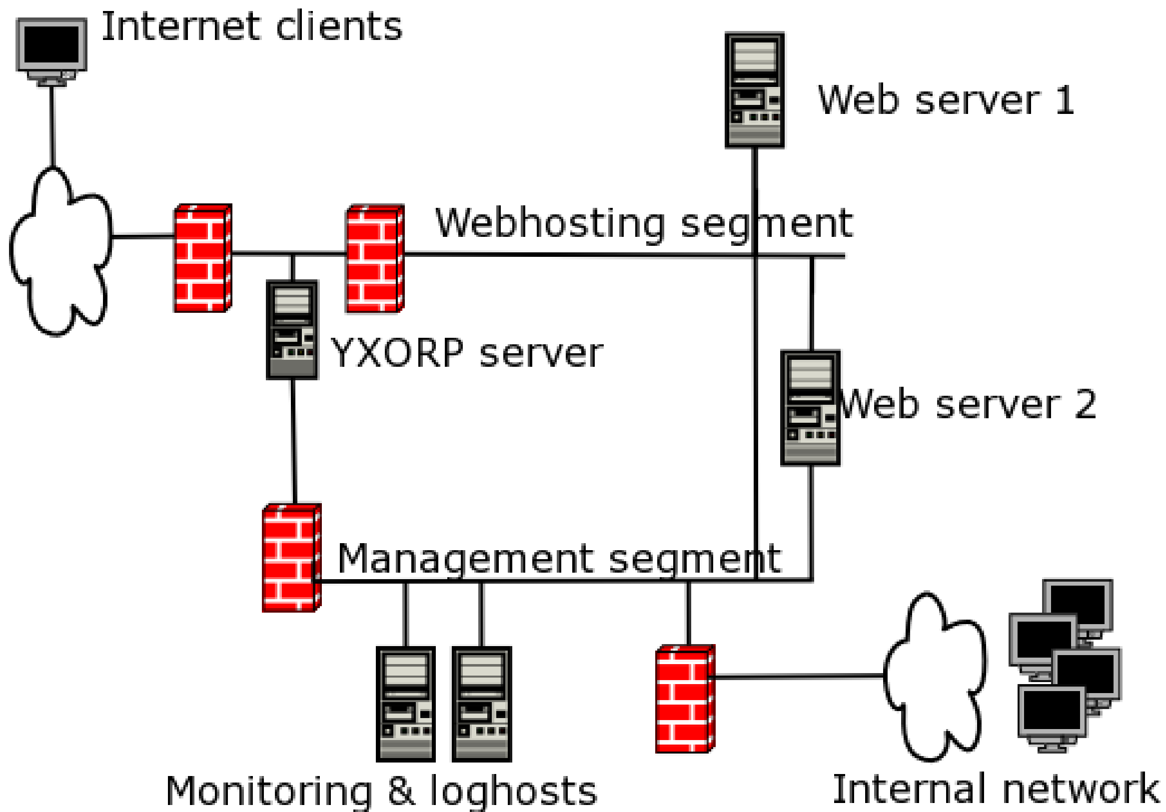
Another example by which almost the same security principles are reached is shown below. It uses a firewall with three interfaces; one connects to the outside world, one connects to the internal network, and one connects to Yxorp.



This setup is almost as secure as the two-firewall setup. The differences are mainly that the complexity of the firewall rules is increased, and that one wrong rule in the firewall could open up the internal network. In a corporate environment, you would want to prevent that; if you are just running Yxorp at home and are in control of all involved systems, the differences are not important.

A better example of a corporate environment is in the next drawing. It shows a two-firewall setup again, but this one is even more complicated because management traffic is separated from production traffic. Reasons for doing this would be like: being able to segregate duties between several systems management teams, having some form of control over the infrastructure even if a denial of service attack is in progress, better insight in traffic flows so network IDS boxes would be easier to configure, etc.

Note that the drawing is simplified; for instance, it does not shows what happens behind the web servers.



Normally, there would be another security zone there for the application servers, and maybe even another one for things like database servers and mainframes.

YXORP box setup

If you want to use Yxorp to enhance security, you must pay attention to the system that Yxorp will run on. Ideally, the system would be hardened (i.e. all security settings should be at the maximum level that will allow the box to do what you need it for) and also it should be minimized (i.e, there should not be any software that you do not need installed on it).

For Linux, most distributions come with a minimum software set; that would be a good start. Take a look at what is included in the software set anyway, some distributions for instance install inetd and telnet. Solaris currently has the reduced networking set, which is appropriate for this kind of system.

The next step is hardening. Take a look at what listening ports there are, and determine if you really need them. Normally, a box running Yxorp only would need sshd to be active, so you can manage the box (you can of course also use a console, but that is rather awkward if you need to upgrade or patch the box). Also look at what processes are running; if there are any you don't need, prevent them from starting or uninstall the software for them.

Domain names

If a user enters http://www.example.com/ in his browser, the browser connects to the IP address that his system thinks www.example.com lives on (i.e. what his DNS server tells his system). So, for Yxorp to do anything meaningful, www.example.com needs to resolve to the IP address that Yxorp runs on.

Yxorp then needs to forward the request to the real server. Yxorp uses it's own mechanism for this, which is based on IP-addresses only (it would not make sense to use DNS for this). However, the web server will need to think it is called www.example.com because otherwise it will not be able to find the right content for the request (at least not if we use HTTP/1.1 and a virtual hosting capable web server).
Exactly what and how you need to configure for this is very much dependent on the actual web servers.

The most important part to remember is that the domain name needs to point to Yxorp, and the Yxorp configuration needs to list the IP-addresses of the real servers.

SSL certificates

If you want to use SSL between the clients and Yxorp, you will need to install keys and certificates on the Yxorp server. The default installation procedure “make install” will generate a test certificate for you automatically.

Normally, using SSL will block virtual hosting. Since the certificate must be sent before the Host: header can be read, it is not possible to send the correct certificate for the domain name in the Host: header. There is a way around this; it is possible to generate certificates containing more than one domain name, but as far as I'm up to date on this, only IE correctly checks for this variant; Mozilla-derivatives, Safari & Konqueror reject the certificate.

For personal use, this is not a bad thing; it only means you will get a popup window asking if you want to accept the certificate. For public use, this means that the users have no way of being sure that the site they are connecting to is the right one.

Basic configuration tutorial

Basic setup

The default installation process (i.e. “configure;make;make install”) installs a configuration that demonstrates some of the things Yxorp can do. There are no complicated things necessary to run this configuration; it only needs a machine to run Yxorp and a machine (possibly the same) that runs browsers and can connect to the Yxorp machine.

The default configuration is also a starting point in this tutorial, which gives an overview of the things Yxorp can do.

This default configuration is described here in detail.

```
<?xml version="1.0"?>
<yxorpconfig>
<!-- Main yxorp configuration file -->
<!-- where the log files go -->

<log accesslog="/var/log/yxorpaccess" errorlog="/var/log/yxorperror" />
```

The first interesting line here is the <log> tag; it sets where the logfiles will go. Log formats are left at their defaults.

```
<!-- This is the tcp port the xml listener will run on -->

<configlistener port="7780" ipaddress="127.0.0.1" />
```

The <configlistener> tag enables the configuration listener. This makes it possible to use online reconfiguration; i.e. you do not have to stop and restart Yxorp every time you make a change in the configuration. Also, the command “yxorpconfig -r” can be used to look at the actual configuration (including all defaults) that is running.

```
<!-- request rule that will check and modify a request -->

<rule id="rule1" type="request">

<![CDATA[
if (Host: ~/^[a-z\.]+$/) {
   Host: = "yxorp.sourceforge.net";
} else if (Host: ~/^[0-9\.]+$/) {
   reject("You are not allowed access through the IP address of this server");
} else {
   Host: = "yxorp.sourceforge.net";
}
]]>

</rule>
```

This <rule> tag defines the rule program that will be executed on receiving a request (to enable this, the rule must be referenced somewhere; usually this is done on the listener, so that any request comes in on the listener causes the rule to be run). The rule looks at the value of the Host: header; if it contains only letters and dots (as in most domain names), it changes the Host: header to “yxorp.sourceforge.net”. If however the Host: header contains numbers and dots (as in IP addresses), it rejects the request.

```
<!--the reject rule will be executed if something is wrong with the request-->
<rule id="rule3" type="reject">
<![CDATA[
errorhtml="<html>
   <head>
   <title>Nicht Touchen, U only watchen das Blinkenlights!</title>
   </head>
   <body>
   <h1>Error ";
errorhtml=concat(errorhtml, errorcode, " - ", rejectreason);
errorhtml=concat(errorhtml, "</h1></body></html>");
Server: = "yxorp-x.x";
]]>
</rule>
```

This <rule> tag defines the reject rule, that is used whenever a request is rejected (either by the Yxorp code, or by the use of the reject function in a script). As with the request rule, it is normally set on the listener. This rule composes a customized error message by using special variables like errorhtml and errorcode. Note how the errorhtml variable is set; the string spans multiple lines.

```

<!-- the definitions for the listeners -->
<listener
   id="test1"
   ipaddress="0.0.0.0"
   port="80"
   rule="rule1"
   rejrule="rule3"
/>
```

This bit defines the listener. Note especially the IP address; as it is set here, this will cause Yxorp to bind to all the available interfaces in the system. If you set a specific IP address, Yxorp will listen only on that address. Also note the rule settings; this sets the linking between the listener and the rules for the requests that come in on this listener.

```
<server id="yxorp.sourceforge.net" virtualserver="group1" />

<virtualserver id="group1" schedule="lru">
   <real id="sf" />
</virtualserver>

<realserver id="sf" ip="dns" inservice="yes" />
```

This bit defines what happens with the contents of the Host: header in a request. This value is first matched to a <server> definition. The one in this example maps to a <virtualserver> group (and will run a sorry rule if none of the servers in this group are available). The virtualserver group defines only one real server (so the schedule setting is superfluous). The last line defines the real server; note that it does not list an IP address but “dns”; this causes the Host: header to be resolved in DNS and contacted. This is useful for examples and testing, but do not use this for real, since it may have unintended consequences i.e. for security and performance. In this case it is very easy, because I don't have to update configurations and documentation if Sourceforge decide to change their IP addresses.

The complete configuration file for this example should have been installed by “make install” with the default configuration file name “yxorpconfig.xml”. you can also pick it up from http://yxorp.sourceforge.net/examples/yxorpconfig.xml

Testing the configuration

Start yxorp. Since release stable-1.23.2, the default is for yxorp to run as a daemon; if you give the command

```
yxorp
```

yxorp will start itself as a daemon process. If you want to stop the daemon, use the command

```
yxorp -K
```

or you can install one of the init.d scripts from the data directory and use your distribution's way of stopping and starting daemons.

If you want to look at lots of debugging output, and have debugging enabled in your build (the default for this is dependent whether or not you downloaded a development or stable release), you can set options like -X -ddddddd; this overrides the configured debug settings to show almost all output. You will need to define where the debug output goes with the <log> tag in the configuration, or run yxorp in the foreground with a command like

```
yxorp -N
```

If you did not change the default installation, and did not set -c <configfilename> as an option to yxorp, the configuration described in the paragraphs above should be active.

Verify it is working by pointing your favorite browser to http://localhost/ ; you should now get the Yxorp website that is really at http://yxorp.sourceforge.net/

Adding SSL

In this step, we will merge the following bits in the active configuration:

```
<?xml version="1.0"?>
<yxorpconfig>
<!-- the definitions for the ssl listener -->
<listener
   id="test1ssl"
   ipaddress="0.0.0.0"
   port="443"
   rule="rule1"
   rejrule="rule3"
   ssl="yes"
   certfile="yxorptest.pem"
/>
</yxorpconfig>
```

As you see above, this configuration snippet defines another listener, but with some added fields for SSL. The most important bit is the “ssl=yes” switch, that determines that this listener will actually expect SSL. The other important bit is the certfile definition; this is the filename that the certificate is stored in. Yxorp will try to read this file from the sysconfdir that has been set in by configure; usually this is something like /usr/local/etc or /etc.

The certificate should have been automatically generated during make install, but you can also use your own certificate.

The complete configuration file for this example is available in the Yxorp source tree in the data/examples directory, it is called add_ssl.xml. You can also pick it up from http://yxorp.sourceforge.net/examples/add_ssl.xml

Testing the configuration

If you stopped Yxorp after the step above, start it again. Apply the extra SSL configuration bits by running

the command “yxorpconfig -c add_ssl.xml”.

Verify it is working by pointing your favorite browser to https://localhost/ . You should get a popup window from your browser about the certificate, since it is self-signed. Depending on your browser, you may also get a warning about nonsecure items on the page, this is because the Sourceforge logo on top of the Yxorp page is inserted with an absolute http URL, and thus is retrieved directly from the original Sourceforge site instead of being proxied by Yxorp.

Adding basic authentication

A simple example of adding basic authentication is in the following configuration snippet:

```
<?xml version="1.0"?>
<yxorpconfig>

<rule id="rule1" type="request">
<![CDATA[
if (Host: ~/^[a-z\.]+$/) {
   Host: = "yxorp.sourceforge.net";
} else if (Host: ~/^[0-9\.]+$/) {
   reject("You are not allowed access through the IP address of this server");
} else {
   Host: = "yxorp.sourceforge.net";
}
if (uri ~/^\/yxorpdoc.*$/) {
   basic_auth_check("my-realm", "local");
}
]]>
</rule>

<basicauth realm=”my-realm” userid="aladdin" passwd="sesame" />

</yxorpconfig>
```

Note that the source for rule1 is completely replaced. In the last if-statement, the regexp checks if the URI contains /yxorpdoc in the beginning; if it does, it will demand basic authentication by the function call to 'basic_auth_check("my-realm", "local")'. Basic_auth_check will reject the request if valid basic authentication credentials are not present in the request; thus, the reject rule we already had will be activated.

The basic authentication credentials that are accepted are defined in

```
<basicauth realm=”my-realm” userid="aladdin" passwd="sesame" />
```

The realm=”my-realm” portion in the definition must correspond to the first parameter in the basic_auth_check call. This allows you to define several “realms”, each of which may give access to different parts of your content. The userid/password combinations are unique within the same value of the realm, but not across realms; so the same userid and password may be defined multiple times in several realms.

Testing the configuration

If you stopped Yxorp after the step above, start it again. Apply the basic authentication configuration bits by running the command “yxorpconfig -c add_basicauth.xml”. Again, the file is in data/examples, or http://yxorp.sourceforge.net/examples/add_basicauth.xml

Verify it is working by pointing your favorite browser to http://localhost/ . This should work as before. Now point your browser at http://localhost/yxorpdoc-2.html and you should get a popup window asking for userid and password.

Now run the command “yxorpconfig -r”. This will read the complete configuration from Yxorp as it is currently active. You will see lots of configuration statements that were not in the files we've seen so far; these are defaults that Yxorp assumes.

Look for the line saying:

```
<basicauth realm=”my-realm” base64="YWxhZGRpbjpzZXNhbWU=" />
```

This is the internal representation that Yxorp uses for basic authentication credentials. Note that this is not any form of encryption (which would not make a lot of sense since basic authentication is not very secure), but the base64-encoded form of userid and password. You can specify either the base64 or userid/password forms in a configuration.

Adding support for non-standard headers

Some sites use other headers than those described in RFC2616 and later; also, Yxorp does not necessarily know about all headers in all RFC's (since there are lots). Yxorp discards headers it does not know about. If however you need one of these headers, you can add it to Yxorp's table of headers using the following example:

```
<?xml version="1.0"?>
<yxorpconfig>
<globalconfiguration>
   <header id="X-Pad:" xlateid="X_Pad:" client="1" server="1" reject="0"
      check="rfc2616-text" maxlen="80" />
</globalconfiguration>
</yxorpconfig>
```

In this example, a header named “X-Pad” is added (this header is at the time of writing sent out by Sourceforge's project web servers, where Yxorp's website resides).

Refer to the configuration web pages for full detail on the settings that you can specify for a header. One thing to look at is the check parameter, which sets the character set that Yxorp will check the contents of the header to, and the reject parameter, which determines what will happen if a header does not conform to what Yxorp thinks it should.

Testing the configuration

Testing the effect of the change above is a bit harder than with the previous examples, since I'm not exactly clear on when Sourceforge's servers insert this header, and some browsers don't allow you to look at the headers. One definite way would be to trace the traffic between your browser and Yxorp with Ethereal, for example.

As before, apply the change by the command yxorpconfig -c add_headers.xml, find the file in data/examples or on http://yxorp.sourceforge.net/examples/add_headers.xml

Adding load balancing

To use load balancing, we need to add a second web server (i.e., a “real server”) to the configuration and setup the “virtualserver” to know about this second real server. How this is done is demonstrated by the following configuration snippet:

```
<?xml version="1.0"?>
<yxorpconfig>

<virtualserver id="group1" schedule="lru">
   <real id="sf" />
   <real id="sf2" />
</virtualserver>

<realserver id="sf" ip="dns" inservice="yes" />
<realserver id="sf2" ip="dns" inservice="yes" />

</yxorpconfig>
```

Note that what this does is just duplicate the definition of the original real server. In a real example, you would obviously define different servers (otherwise, what's the point of load balancing), and use IP addresses instead of the “dns” testing shortcut.

Testing the configuration

Apply the change: yxorpconfig -c add_loadbalancing.xml from data/examples or http://yxorp.sourceforge.net/examples/add_loadbalancing.xml and access the site a couple of times via http://localhost/ . Then look at the access log file (usually in /var/log/yxorpaccess) and look for the realserver names in the log entries, these should look similar to these:

```
127.0.0.1 27340 [09/Jan/2006:17:39:17 +0100] sf "GET
http://yxorp.sourceforge.net/yxorpdoc-2_html_m34ac0a87.png" 200 501 15874 546
15836 501 299 546 261 737

127.0.0.1 27342 [09/Jan/2006:17:39:17 +0100] sf2 "GET
http://yxorp.sourceforge.net/yxorpdoc-2_html_m64bcb189.png" 200 501 17568 546
17530 501 299 546 261 753

127.0.0.1 27346 [09/Jan/2006:17:39:17 +0100] sf2 "GET
http://yxorp.sourceforge.net/yxorpdoc-2_html_1cb9e497.png" 200 500 17250 545
17212 500 299 545 261 744

127.0.0.1 27344 [09/Jan/2006:17:39:17 +0100] sf "GET
http://yxorp.sourceforge.net/yxorpdoc-2_html_m5fd42.png" 200 498 32706 543
32668 498 299 543 261 945
```

As you see, both realservers are scheduled.

You can now have a look at the realserver status with the yxorprealserver command, as follows:

```
# yxorprealserver -v
sf          : inservice        available        config
sf2         : inservice        available        config
#
```

Other options on the yxorprealserver command allow you to manually set a server out-of-service or unavailable:

```
# yxorprealserver -v -u sf
sf          : inservice        unavailable      config
sf2         : inservice        available        config
#
```

Note the difference between inservice and available; Yxorp can automatically set a realserver out-of-service if it fails to respond to a connection several times in a row (and Yxorp will also automatically set it inservice again if it responds to a new connection that is scheduled by the auto-wakeup mechanism). Yxorp will however not change the available/unavailable flag.

There is another difference between inservice and available, this has to do with sticky load balancing. See the example on that for a discussion.

One last thing to try with this setup is what happens if you set both realservers out-of-service. Note the ugly message a client gets in this case. This is where the sorry rule comes in.

Adding a sorry rule

If no servers in a virtualserver group are available, it is possible to let a sorry rule run. A sorry rule can generate a custom HTML page, as we did above with the reject rule. However, especially with a complex layout for a sorry page, this can be a lot of work, and another technique is possible that we will see in the example below:

```
<?xml version="1.0"?>
<yxorpconfig>

<rule id="sorryrule" type="sorry">
   redirect("http://yxorp.sourceforge.net/noserver.html");
</rule>

<server id="yxorp.sourceforge.net" virtualserver="group1" sorry="sorryrule" />

</yxorpconfig>
```

The sorry rule just generates a redirect (i.e., a 307 Temporary Redirect status code, with a Location: header set to http://yxorp.sourceforge.net/noserver.html.

Also note that the definition of the sorryrule is on the server level, not on the virtualserver. This is because it may be necessary to have a different sorry page for each server definition, even though these servers may share the virtualserver group.

Testing the configuration

As above, activate the file add_sorryrule.xml from data/examples or http://yxorp.sourceforge.net/examples/add_sorryrule.xml, then make sure both realservers are set out-of-service or unavailable, and then try http://localhost/ and if all has gone well, you should now be redirected to http://yxorp.sourceforge.net/noserver.html.

Sticky load balancing

Sticky load balancing is the technique where many clients are distributed evenly over a group of servers, but each request from a specific client is always sent to the same server, i.e. the load balancer “remembers” the association between a client and a server. This is necessary if an application runs on the webservers that is not stateless, and there is no other means of sharing the state over the servers. If you want to share state on the web/application server level, or if you want sticky load balancing, is a discussion that goes rather far beyond the scope of this tutorial.

Yxorp can do sticky load balancing, however, the implementation has some drawbacks compared to straight load balancing; at least, the total throughput of Yxorp will decrease somewhat. Also, the table in which the state mapping is maintained may grow large, especially if someone is doing denial-of-service attacks on your site (or worse, targeted attacks tailored to let Yxorp's tables grow).

The way Yxorp keeps track of which client is which is by setting a session cookie. This cookie will normally be named “yxorpstate”, and will be valid until the end of the “session” (in most cases, this is the lifetime of the browser instance). The cookie value will be a randomly generated string like XL61DS0FPS4PLEDXUIBVIB8LLU301IZX1ZOB1WVWSOW9SWNMG2CEXGZFW6CQ1L.

The configuration changes to enable sticky load balancing are simple:

```
<?xml version="1.0"?>
<yxorpconfig>

<virtualserver id="group1" sticky="yes">
   <real id="sf" />
   <real id="sf2" />
</virtualserver>

</yxorpconfig>
```

In fact, the only change this makes is the “sticky=yes” part.

Testing the configuration

Activate the configuration from http://yxorp.sourceforge.net/examples/add_sticky.xml or data/examples. Then, point your browser to http://localhost/ and cause a couple of hits.

Then, use the yxorpclientstate command as follows:

```
# yxorpclientstate
<yxorpclientstate>

<clientstate
   id="IQ3AHIIPFKEDO1SQUQ1H67CPX68NMFSUARUWQESXGEXGGQT9TU74DJYU92C7V0"
   clientip="127.0.0.1" sticky="1" lastactive="[09/Jan/2006:23:52:37 +0100]"
   hitcount="1" toclient="9850" fromclient="451" toserver="9850"
   fromserver="451" >

<stickymap server="yxorp.sourceforge.net" realserver=sf2" />

</clientstate>

<tablestats entries="1" tablebytesize="824" maxchainlength="1"
   tabletruncated="0" />

</yxorpclientstate>
#
```

As you see, the value of the state cookie is linked to a lot of information. For this example, note the stickymap tag; this links requests for a certain server to a specific realserver.

This gets more complicated if you have more servers defined. Normally, the clients (browsers) will get a different state cookie for each server (or, rather, what the client thinks is a server). It is possible to use the same cookie for several servers by putting the servers in a cookie domain, see the setcookiedomain function call for details on this. If cookiedomains are used, the same cookie and state can be used for the servers sharing the domain (in this case you would see multiple stickymap tags in the same clientstate tag).

Client states are cleaned up automatically by Yxorp, see the global configuration settings.

Sticky loss rules

Similar to the sorry rules we already saw, there is also the case in sticky load balancing when the mapped server is no longer available. In this case, Yxorp has a special rule called “stickyloss” that handles what happens with the request.

```
<?xml version="1.0"?>
<yxorpconfig>

<rule id="stickylossrule" type="stickyloss">
   redirect("http://yxorp.sourceforge.net/stickyloss.html");
</rule>

<server id="yxorp.sourceforge.net" stickyloss="stickylossrule" />

</yxorpconfig>
```

You would typically want a stickyloss rule to explain that clients have lost their state in an application, and need to retry to another server.

Testing the configuration

Activate the configuration from http://yxorp.sourceforge.net/examples/add_stickyloss.xml or data/examples. Use the yxorprealserver command to check if you need to restore the realservers to available and inservice after the experiments in the previous example. Then, point your browser to http://localhost/ and cause a couple of hits.

Then, use the yxorpclientstate command as follows:

```
# yxorpclientstate
<yxorpclientstate>

<clientstate
   id="PQJZNM51ELUXTKDNECCZQIRLC1ME0BZBALX68AWBLMKIE978NV22YEXWLWMKV4"
   clientip="127.0.0.1" sticky="1" lastactive="[10/Jan/2006:00:35:36 +0100]"
   hitcount="3" toclient="29584" fromclient="1763" toserver="29584"
   fromserver="1763" >

<stickymap server="yxorp.sourceforge.net" realserver=sf" />

</clientstate>

<tablestats entries="1" tablebytesize="824" maxchainlength="1"
   tabletruncated="0" />

</yxorpclientstate>
#
```

Note the server that is mapped (in this case it is sf), and set it out-of service as follows:

```
# yxorprealserver -v -o sf
sf          : out-of-service   available        config
sf2         : inservice        available        config
#
```

and see what happens if you reload your browser (remember, it needs to be the same window as before, since Yxorp's state cookie should only be valid in a single browser instance). If everything is correct, nothing apparently has changed. Why? Because we just set the sf server out-of-service; it will not be eligible for new sticky mappings, or normal scheduling, but it will continue to serve for valid states.

Next, to see the stickylossrule in action, we will do:

```
# yxorprealserver -v -u sf
sf          : out-of-service   unavailable      config
sf2         : inservice        available        config
#
```

If you reload your browser again, you should now see the effect of the stickyloss rule.

Rule progamming examples

This chapter deals with using rules for more advanced reverse proxying. The examples given are only parts of a complete configuration. In most cases, an example of a rule is shown, to illustrate the subject of the paragraph. These rules may however not be complete; you should not just copy them and think your web server is secure. Remember, a reverse proxy is a complicated thing, and a minor mistake in the configuration may decrease your security instead of increasing it.

Enforcing SSL for certain requests

If you have a webserver with some unimportant content that you want to allow anyone access to (for instance, pictures of your pets), and also some restricted content (like your webmail application) you may want to make sure that logging in to your webmail server is only possible if SSL is used on the session (and make sure your userid and password will be encrypted on the Internet). This can be done with a rule as follows:

```
<rule id="rule1" type="request">

<![CDATA[
if (uri -/^\/webmail/) {        // if uri starts with “webmail” in lower case
   if (!issslsession()) {
      reject(“Sorry, you're not using SSL”);
   }
}
]]>

</rule>
```

A more userfriendly way is also possible, in which the client is automatically redirected to the SSL variant of the URL that was attempted:

```
<rule id="rule1" type="request">

<![CDATA[
if (uri -/^\/webmail/) {        // if uri starts with “webmail” in lower case
   if (!issslsession()) {
      redirect(concat(“https://”, “www.yourdomain.whereever”, uri));
   }
}
]]>

</rule>
```

SSL to backend servers

Normally, Yxorp will use HTTP to backend servers (i.e. the servers that Yxorp is reverse-proxying for). In some cases however, it is desirable to connect to a backend server (i.e. a server that Yxorp is reverse-proxying for) using SSL.

To do this, some extra definitions are needed for the realserver objects, and a request rule must be used to enable SSL on the backend connection for the requests.

If your realserver definition is like:

```
<realserver id="sf" ip="192.0.2.1" port=”80” inservice="yes" />
```

change it to include the sslport tag as follows:

```
<realserver id="sf" ip="192.0.2.1" port="80" sslport="443" inservice="yes" />
```

Then, in your request rule, add a call to the function "setsslbackend()" as in the following example:

```
<rule id="rule1" type="request">

<![CDATA[

... code to determine if you want to reject the request straight away

if (... some code to see if you want to enable ssl backend encryption for this request) {
   setsslbackend();
}

]]>

</rule>
```

In the current versions, Yxorp does not check if the certificate that the backend server presents is valid. I.e., it could be a self-signed certificate, or it could be out of date, or it could be signed by anyone instead of a trusted certificate authority. In short, you should not use this if you are not in complete control of the backend servers.

Note that the setsslbackend function must be called for each request that you want to use backend SSL for.

Filtering on client IP address, range, or domain name

To allow access to some set of web resources based on the IP address of the client, use one of the examples below as a starting point.

Filtering on a single IP address

It is possible to filter on a single IP address as follows:

```
<rule id="rule1" type="request">

<![CDATA[

if (uri -/some part of the uri you want to filter on/) {
   if (!(getclientip() -/^127\.0\.0\.1$/)) {
      reject("sorry, you don't have the right IP address");
   }
}

]]>

</rule>
```

Note that the IP address is returned from the getclientip() function in its normalized textual form for the protocol in use (IPv4 or IPv6). In the example, the local host would have the address *127.0.0.1* for IPv4, but in IPv6 *::1*, and *::ffff:127.0.0.1* may both occur.

Filtering on an IP range

It is possible to filter on a range of IP addresses as follows:

```
<rule id="rule1" type="request">

<![CDATA[

if (uri -/some part of the uri you want to filter on/) {
   if (!(clientinrange("192.0.2.0/24"))) {
      reject("sorry, you don't have the right IP address");
   }
}

]]>

</rule>
```

The clientinrange() function determines if the client IP address is in the given range. The function can only process IPv4 ranges, however, there is an equivalent IPv6 function called clientinip6range().

Filtering on domain names

Filtering on client domain names can be done as well. The next example shows how to filter on the last two parts of the domain name:

```
<rule id="rule1" type="request">

<![CDATA[

if (uri -/some part of the uri you want to filter on/) {
   if (!(getclientdomainname() -/.*example\.com$/)) {
      reject("sorry, you don't have the right domain name");
   }
}

]]>

</rule>
```

The getclientdomainname() function returns the reverse DNS lookup for the client IP address. Note that in some cases, the reverse lookup of an address gives a different name than a normal lookup. Also, because Yxorp does not normally resolve the names of clients, using this function may impact performance.

Using black-, white- and greylisting

In filtering requests, a blacklist is used to specify what is not allowed. Everything that is not in the blacklist is allowed. In contrast, a whitelist specifies what is allowed; if something is not explicitly in the whitelist, it is not allowed.

Greylisting is a less well defined term, that has lately been associated with a technique for spam filtering. In the Yxorp context, we will use the term to define either black- or whitelists with exceptions for subsets of a list case.

Blacklisting

A typical blacklist rule is shown in the following example:

```
<rule id="rule1" type="request">

<![CDATA[

blacklist uri {
   -/\.exe/
   -/\.dll/
   -/^\/cgi-bin/
} if-failed {
   reject("sorry, you did not pass the blacklist on this server");
}

]]>

</rule>
```

In the example above, all requests containing the regular expressions anywhere in the URI will be rejected., except for the cgi-bin entry, which is anchored to the beginning of the URI.

In general, build your filters to anchor from the start of the URI; otherwise they may be too general (like the .exe and .dll patterns above). Don't anchor patterns only at the end, because webservers will accept several kinds of "parameters" to be passed in an URI (like: "http://localhost/?blablabla", "http://localhost/#somereference" etc). For a blacklist, this does not need to be a problem, since matching too many possible URI's only causes a possibly valid request to be rejected. However, with whitelists this may mean that your filter is not secure.

Whitelisting

A typical whitelist rule is shown in the following example:

```
<rule id="rule1" type="request">

<![CDATA[

whitelist uri {
   -/^\/$/
   -/^\/index\.html$/
   -/^\/pictures/\.*\.jpg$/
   -/^\/htmlfiles\/.*\.html$/
} if-failed {
   reject("sorry, you did not pass the whitelist on this server");
}

]]>

</rule>
```

In the example above, all requests *not* containing one of the regular expressions will be rejected. Note that the expressions are anchored at the beginning of the URI, to avoid the issues with parameters described above. Still, the URI "http://localhost/pictures/hacker.dll?whatever.jpg" would pass this whitelist.

So, how can you make a secure filter? Consider all of these measures:

1. Be careful about what is on your web servers, where it is, and what it is. If you don't need it, it should not be on the server. Especially take care that updates, fixes, and patches do not (re-)install default content, samples, etc.

2. Filter first using a blacklist, and in this way exclude all URI's that you do not want to make accessible. Things like executables for active content (cgi-bin, .php, .dll etc) can easily be excluded in this way.
3. Then use a whitelist, and specify what you do want to make accessible. The whitelist is the most tricky part, because an expression that allows too much will give access to more resources than it should.
4. Test the filter to see if it does what you expect!

Greylisting

Most blacklists and whitelists have a disadvantage: The filtering they provide is in some cases too black-and-white (no pun intended... I mean too all-or-nothing). For instance, in the whitelist example above, all files in the pictures directory are accessible. What if you have several directories there, but want to exclude one from the whitelist? This is where greylisting comes in: a whitelist (or blacklist) with exceptions. The following example shows this:

```
<rule id="rule1" type="request">

<![CDATA[

whitelist uri {
   -/^\/$/
   -/^\/index\.html$/
   -/^\/pictures\/\.*\.jpg$/ : if (uri -/^\/pictures\/secret/)) continue-list;
   -/^\/htmlfiles\/.*\.html$/
} if-failed {
   reject(“sorry, you did not pass the whitelist on this server”);
}

]]>

</rule>
```

The if-statement following the pictures regexp in this example examines the URI more closely; if the regexp in the if-statement matches, the continue-list statement causes the earlier match to be ignored.

This example uses a simple if-statement, however, all normal statements can be used within a white- or blacklist in this way. It is even possible to nest white- and blacklists, as shown in the following example:

```
<rule id="rule1" type="request">

<![CDATA[

whitelist uri {
   -/^\/$/
   -/^\/index\.html$/
   -/^\/pictures\/\.*\.jpg$/ : blacklist uri {
                                  -/^\.pictures\/secret/
                               } if-failed {
                                  reject(“sorry, your request is an exception
to the whitelist”);
                               }
   -/^\/htmlfiles\/.*\.html$/
} if-failed {
   reject(“sorry, you did not pass the whitelist on this server”);
}

]]>

</rule>
```

Using an application to authenticate access to other URI's

Situation

Consider the following situation: You have an existing webmail program on your server (we'll use Squirrelmail in this example). You also have some other stuff on your server, that you do not want everyone to be able to see. Thirdly, you also have public content that anyone may see.

What Yxorp can do with this is using the login to Squirrelmail to authenticate users for the private pages; so, the private pages would only be accessible *after* a user has successfully logged in to Squirrelmail.

So, how does this work? Squirrelmail has a login page on which a user types userid and password. If the user clicks the login button (or presses enter), the browser will post the content (userid & password) to another URI, which is a Squirrelmail page that will verify userid and password. If successful, Squirrelmail will send a 302 “Found” status, with a redirect to the main webmail page, and set an authentication cookie (“key”). If unsuccesful, Squirrelmail will send a redirect to a page with a message that userid and password were invalid.

Yxorp needs to do the following to be able to use this:

1. Track the client, i.e. setup a state cookie.
2. Allow free access to the Squirrelmail login pages.
3. Monitor the returned traffic from Squirrelmail to see if the authentication cookie is present
4. Allow access to other pages based on if the authentication cookie was seen previously.

Note that the authentication cookie will only be sent once, as the result of the post. So, Yxorp must remember it. To do so, we will use the client state variable feature in Yxorp.

Example

The rule that processes requests coming in from clients is as follows:

```
<rule id="rule1" type="request">

// these uri will be served without authentication

whitelist uri {
   -/^/squirrelmail\/src\/login\.php/          // login page
   -/^/squirrelmail\/images\/sm_logo\.png/     // logo on login page
   -/^/squirrelmail\/src\/redirect\.php/)      // this processes the login

// add other public uri here

} if-failed {
   if (!getclientstatedvar("authenticated")) {
      redirect("http://www.example.com/your-home-page");
      return;
   }
}

</rule>
```

In this example, note that the regexp patterns for the URI's are anchored at the start to prevent accidentally matching something else. The final if-statement in the example shows what to do if we have not (yet) seen a valid login; the most user-friendly action here is to send a redirect to your homepage or something, but in general you should not do this as a default action for all requests to your website, because this may give out unnecessary information to hackers. Since anything that is handled here should be a request to something the user is not authorized for, it may be a better idea to reject the request outright.

The second rule we need is the rule that processes the response coming back from the server. It is shown in the following example:

```
<rule id="rule2" type="response">
<![CDATA[
foreach i (enumerate_dupvar("Set_Cookie:")) {
   if (statuscode==302 && $i -/key=/) {
      if (uri -/^/squirrelmail\/src\/redirect\.php/) {
         trace("authenticated");
         setclientstatedvar("authenticated", "true");
      }
   } else if ($i -/key=deleted/) {
      trace("authentication cancelled");
      setclientstatedvar("authenticated", "");
   }
}
]]>
</rule>
```

In this rule example, note the use of duplicate variables and the foreach statement to process all of the variables in a dupvar group. This is necessary because more than one Set_Cookie header may be present in the response from the server, and the one we need (the one that sets “key=”) may not be the first one.

Also note the check for statuscode=302 (“Found”), which is the normal result from a POST request. In contrast, the else part of the if statement does not check for a statuscode, but just for the presence of a Set_Cookie header with a value of “key=deleted”. This is what happens if the user clicks the logout link within Squirrelmail; in this way the key value for the authentication is removed from the browser.

Especially note the explicit check for the URI causing the 302. If it was omitted, any active resource on the server could cause a 302 status code and a Set_Cookie, thereby potentially bypassing the userid and password check that we assume Squirrelmail to do for us.

Finally, the call setclientstatedvar("authenticated", ...) is used to set a client state variable called "authenticated" to the appropriate value. This is the variable we used in the request rule to determine if the request to the private pages are allowed.

Conclusion

So, what we have accomplished is a "login" to some arbitrary pages containing private content, without changing anything on the server. If the rules are carefully coded, and the application you use is solid, this can be a very secure way to control access.

However, you should understand that even though the authentication is performed, Yxorp does not know *which* user is logged on. As such, this example may be used to grant access to users, but it can not be used to give different access to different users.

Configuration reference

Yxorp reads its configuration from an XML file at startup, and sets up it's internal representation of the configuration statements. While Yxorp is running, a new configuration file can be inserted into the running configuration; this may extend the configuration, but also replace parts of the original configuration. It is also possible to read the actual configuration from the running daemon.

Overview of the configuration file

The configuration statements are grouped; each group deals with a set of configuration items. These groups are:

- configlistener; this group deals with the TCP interface that is used to read or update the configuration in an active yxorp daemon;
- log; this group contains the statements that configure logging;
- listener; this group, that may occur more than once in a configuration, controls the proxy processing and sets the TCP ports that the yxorp daemon will listen to;
- rule; may occur multiple times and contains the rule programming language statements;
- threads; controls the settings that the yxorp daemon will use for worker threads;
- security; controls security settings;
- server; controls which (virtual or real) servers yxorp can send requests to;
- virtualserver; controls the groupings of servers that are used as loadbalancing or failover pools;
- realserver; controls the settings of real servers that are mapped from a server definition or a virtualserver pool;
- globalconfiguration; controls which settings the yxorp daemon uses for things like timers, retry limits to connect to servers, headers to process, etc;
- basicauth; sets which userid/password combinations are recognized for local basic authentication processing;
- debug; controls filtering of debug output messages. If yxorp is built with debugging support enabled, these filters may be used to selectively show some message types.

Syntax of the configuration file

The configuration file should be formatted as follows:

```
<?xml version="1.0"?>
<yxorpconfig>
<log>
... logging settings
</log>
<configlistener>
... config listener settings
</configlistener>
<rule id=”name” type=”request”>
... rule code
</rule
</yxorpconfig>
```

Configuration file contents

The configuration file must start as follows:

```
<?xml version="1.0"?>
<yxorpconfig>
```

and end as follows:

```
</yxorpconfig>
```

The other contents of the configuration file depend on what Yxorp functions you want to use. These are described in the next sections.

Configuring listeners

Listeners are the end point for sessions between a client and Yxorp. For Yxorp to do any useful work, at least one listener must be created.

There are two types of listener, corresponding to the two types of proxy: dissecting and tunneling. The dissecting proxy completely takes a request apart, may run all kinds of scripts on each part of the request, and then reassembles it and forwards it to a server. In contrast, a tunnel proxy does not do any processing on a request, but just forwards it as-is to a server. Which type of proxy is chosen depends on the definition in the listener; a listener may invoke either type of proxy, but not both.

A listener is created by setting a <listener> tag in the configuration. The <listener> tag takes the following attributes:

Attribute	*Format*	
id	string	Must be set on each listener, and sets the listener identifier. The id must be unique. It is used in logging and tracking, and in the reconfiguration process.
type	number	4 means IPv4, 6 means IPv6. Default is IPv4.
mode	string	“tunnel” creates a tunnel proxy; “dissect” creates a dissecting proxy. “dissect” is default.
port	number	TCP port number the listener will bind to.
ipaddress	ipaddress	IP address the listener will bind to. If type=4, then a normal dotted IP address must be set; if type=6, an IPv6 address notation must be set. The IPv4 address 0.0.0.0 has a special meaning and will bind to all local addresses of the system.
rule	string	The name of a rule. If this listener has mode=”dissect”, the rule will be

Attribute	Format	
		executed for requests; if mode=”tunnel”, this attribute has no meaning.
resprule	string	The name of a rule. If this listener has mode=”dissect”, the rule will be executed for responses; if mode=”tunnel”, this attribute has no meaning.
rejrule	string	The name of a rule. If this listener has mode=”dissect”, the rule will be executed for rejected requests; if mode=”tunnel”, this attribute has no meaning.
tunneldest	ipaddress	If mode=”tunnel”, IPv4 address of the tunnel destination host; if mode=”dissect”, this attribute has no meaning.
tunnelport	ipaddress	If mode=”tunnel”, TCP port number on the tunnel destination host; if mode=”dissect”, this attribute has no meaning.
ssl	string	“yes” to enable SSL, “no” to disable. If Yxorp is built without SSL support, presence of this attribute will cause an error message.
certfile	string	Name of a file containing the certificate to be used. The certificate must be in PEM format.
certpasswd	string	Optional password to decrypt the certificate with
sslcachedisable	number	Used for testing only, don't use in any kind of serious setup
ssldebug	number	Used for testing only, don't use in any kind of serious setup

Reconfiguring

All attributes on a listener can be reconfigured on-line. If the socket attributes (port and ipaddress) are changed, Yxorp will close the current socket and bind to a new socket with the new settings. If the new socket is in use on the operating system level, this might take an extended period of time; if sessions or programs are active on the new socket, the socket may not become available at all, causing Yxorp to keep retrying to bind to the socket.

Example

The following example creates a dissect listener for IPv4 on the HTTP port, that will bind to the localhost interface (and thus only accept requests sent to the localhost address):

```
<listener
        id=”www”
        type=”4”
        mode=”dissect”
        ipaddress=”127.0.0.1”
        port=”80”
        rule=”wwwreq”
        resprule=”wwwresp”
        rejrule=”wwwrej”
/>
```

The next example will create a tunnel listener using IPv6 on the HTTPS port, and connect to an IPv4 server on address 1.2.3.4:

```
<listener
              id=”www”
              type=”6”
              mode=”tunnel”
              ipaddress=”fe80::210:1100:0123:4567”
              port=”443”
              tunneldest=”1.2.3.4”
              tunnelport=”443”
/>
```

Configuring servers

Yxorp uses the servers listed in the configuration as an abstraction layer for servers. A server can either directly map to a real server, or map to a virtual server for load balancing. If no servers are available, a special rule type called the 'sorry rule' is executed.

A server is identified by the name taken from the Host: header, and thus, normally the same as the domain part of the URL that was requested. If rule code is used to modify the Host: header value, the changed value is used. If the value in the Host: header does not correspond to any server, the request is rejected (by executing a reject rule, if one is specified; or by the default reject actions).

A server is created by setting a <server> tag in the configuration. The <server> tag takes the following attributes:

Attribute	***Format***	
id	string	Must be set on each server. The id must be unique. The id is used to select a server based on the contents of the Host: header.
virtualserver	string	Identifies the virtual server that will process the request. Mutually exclusive with the realserver attribute.
realserver	string	Sets a direct mapping from server to real server. Mutually exclusive with the virtualserver attribute.
sorry	string	Name of the rule to be executed if no real servers are available for processing the request.
stickyloss	string	Name of the rule to be executed if the real server a sticky session was mapped to is no longer available.
track	string	“yes” or “no” to indicate if requests to this server cause tracking to be invoked for this client.

Reconfiguring

All attributes on a server can be reconfigured on-line.

Example

The following example creates a server that will map requests for a domain name (Host: header value) of yxorp.sourceforge.net to a virtual server named “servergroup1”. If none of the servers in the group is available, the sorry rule “sorryrule-yxorp” is invoked.

```
<server
              id=”yxorp.sourceforge.net”
              virtualserver=”servergroup1”
              sorry=”sorryrule-yxorp”
/>
```

The following example creates a server that will map requests for a domain name (Host: header value) of

yxorp.sourceforge.net to a real server named “bigserver”:

```
<server
            id="yxorp.sourceforge.net"
            realserver="bigserver"
            sorry="sorryrule-yxorp"
/>
```

Configuring virtual servers

The configuration settings for a virtual server list which real servers may be used to process a request. The virtualserver configuration also determines the scheduling algorithm to be used for load balancing.

Real servers will be excluded from the load balancing if they are out-of-service. If one of the real servers has been set out-of-service automatically, is is possible to have the virtualserver schedule a request to it occasionally, to determine whether it has become available again. This mechanism is called “real server wakeup”. This should work without disruption; if Yxorp is not able to connect to a server, another server will be connected after a short timeout.

A virtual server is created by setting a <virtualserver> tag in the configuration. Inside the virtualserver tag , the <real> tags are used to list the real servers. The <virtualserver> tag takes the following attributes:

Attribute	***Format***	
id	string	Must be set on each server. The id must be unique.
schedule	string	Identifies the scheduling algorithm the virtual server will use. Currently defined are “roundrobin”, “randomrobin” and “lru”.
wakeup	string	Enables or disables waking up of mapped real serves that have been automatically set out-of-service. Valid values are “auto” for automatic wakeup and “manual” for disabling the mechanism.
wakeupfrequency	number	Average number of requests scheduled to in-service real servers in between attempts to wakeup a real server. Do not set to a value less than 100.
sticky	string	“yes” or “no” to set sticky load balancing for this client on this server group.
mode	string	“add”, “delete” or “clear”; values pertain to the actions that should be taken on <real> tags enclosed in this virtual definition. If add, <real> servers will be added to the list; if clear, all <real> servers will be deleted before adding the <real> servers in the virtual definition; if delete, listed <real> in the virtual definition will be deleted. Default is clear, so that the definition you include in the virtualserver tags will completely replace all current definitions in the daemon.

The <real> tag takes the following attributes:

Attribute	***Format***	
id	string	Must be set on each server. The id must be unique.

Reconfiguring

All attributes on a virtualserver can be reconfigured on-line. The list of real servers will by default be replaced.

Example

The following example creates a virtual server that will load balance requests over a group of three real

servers, using the roundrobin scheduling algorithm, and automatically retrying to inservice servers that may have become unavailable:

```
<virtualserver
             id="servergroup1"
             schedule="roundrobin"
             wakeup="auto"
             wakeupfrequency="100"
 >
             <real id="server1" />
             <real id="server2" />
             <real id="server3" />
</virtualserver>
```

Configuring real servers

The configuration settings for a real server define the attributes of actual servers.

A real server may be set out-of-service automatically if it fails to respond to requests.

A real server is created by setting a <realserver> tag in the configuration. The <realserver> tag takes the following attributes:

Attribute	***Format***	
id	string	Must be set on each server. The id must be unique.
ipaddress	IPv4 address	Numerical IPv4 address of this server. It is possible to set "dns" instead of the numerical IP address here; this will cause the value of the Host: header to be used to lookup the IP address and connect to it. This is useful for debugging and testing, but do not use this in production situations, as it might be a security risk, and it will also decrease performance. Note also that "dns" only works for requests associated with a dissecting listener, not with a tunnel.
port	number	TCP port on the server . If not set, 80 is the default.
sslport	number	SSL-capable TCP port on the server.
available	string	"yes" or "no" to set the intended availability state
inservice	string	Possible values are "yes" or "no"; automatically changed by server wakeup mechanisms (see virtualserver).
reason	string	(display only) The source of the information in the inservice attribute; "config" if the setting was derived from configuration file or command; "auto" if the setting was caused by an automatic process.

Reconfiguring

All attributes on a real server can be reconfigured on-line.

Example

The following example creates a real server. The initial state of the real server is inservice, that is, the server is presumed to be able to process requests; the server is also set available, so it will immediately be scheduled.

```
<realserver
             id="bigserver"
             port="80"
             ipaddress="10.1.0.117"
             inservice="yes"
             available="yes"
/>
```

Configuring rules

Rules are small programs written in a special programming language. Each rule contains exactly one such program, which can deal with a request or response. The rule language itself is described in a separate chapter.

A rule begins with a <rule> tag, and ends with a </rule> tag. The text space in between these tags are assumed to be the program source. The rule program source is compiled when the configuration file is read; syntax errors will be reported. The configuration process will abort if serious errors are detected.

Since the configuration file uses XML, and the rule language uses text constructs that may not be valid in "plain" XML, it is sensible to enclose all rules in <![CDATA[...]]> tags.

The <rule> tag takes the following attributes:

Attribute	***Format***	
id	string	Required; must be unique across a configuration. The listener rule and resprule attributes refer to this id.
type	string	Value can be "request", "response", "reject", or "sorry". The value is currently used for configuration file documentation only; defaults to "request".

Reconfiguring

All parts of a rule may be reconfigured.

Example

```
<rule id="example" type="request">
<![CDATA[
// rule source
   host: = "yxorp.sourceforge.net";
]]>
</rule>
```

Configuring logging

Yxorp logs to files. Normal requests are logged to the access log; requests denied (either by a rule or by Yxorp code) are logged to the error log; and rule programs may log information to another log file, called the trace log. Debugging messages may either be logged to stdout, or to a file.

The log file names are set as attributes on the <log> tag in the configuration file.

The <log> tag takes the following attributes:

Attribute	***Format***	
accesslog	string	File name for the access log
debuglog	string	File name for the debug log. Note that you only need this if Yxorp has been built with debugging enabled, and if debug settings in the

Attribute	*Format*	
		configuration actually cause debugging output. If no file name has been set, and debug output is generated, it will be sent to stdout.
errorlog	string	File name for the error log
tracelog	string	File name for the trace log. You will only need this if you have rule programs containing the trace function.

Log entries in the access and error logs are formatted from a printf-like format string. Any normal text in the format string is simply copied to the log; tokens starting with '%' are interpreted as field names, and substituted with the field value. The format strings are set as subtags enclosed in the log tag; the format strings are set in the text space. The format for the access log is set using the <access-fmt> subtag; the format of the error log is set using the <error-fmt> subtag.

The default access log format is as follows:

```
%clientip %clientport %nsstart %realserver \"%method %protocol%host%uri\"
%statuscode %fromclient %toclient %toserver %fromserver %clientreqhdr
%clientrsphdr %serverreqhdr %serverrsphdr %elapsed
```

The default error log format:

```
%requestnumber %clientip %clientiplookup %clientport %nsstart %realserver
\"%method http://%host%uri\" %statuscode %fromclient %toclient %toserver
%fromserver %clientreqhdr %clientrsphdr %serverreqhdr %serverrsphdr %elapsed
%rejectreason
```

The following tokens are available for formatting:

Token	*Field value*
%clientip	Client IP number
%clientiplookup	Resolved name of client IP number
%clientport	TCP port number on client
%method	Method name
%uri	Uri (after translation by rule programs)
%host	Host name of server (after translation by rule programs)
%nsstart	Time stamp at start of request
%nsend	Time stamp at end of request
%logi	Ordinal number of log entry as counted by request logging module
%requestnumber	Ordinal number of request, as counted by listener at start of the request
%protocol	Returns "http://" or "https://" according to the current protocol
%statuscode	HTTP status code
%fromclient	Total bytes received from client
%toclient	Total bytes sent to client
%fromserver	Total bytes received from server
%toserver	Total bytes sent to server
%clientreqhdr	Number of bytes in headers received from client
%clientrsphdr	Number of bytes in headers sent to client
%serverreqhdr	Number of bytes in headers sent to server
%serverrsphdr	Number of bytes in headers received from server

Token	Field value
%elapsed	Wall clock time it took to process the request, in milliseconds
%rejectreason	Text string containing the reason why a request was rejected
%realserver	The real server that the request was forwarded to

Reconfiguring

All of the log settings may be reconfigured. If the log file names are changed in the new configuration, the active log file will be closed, and a new file with the changed name will be created (as yxorp tries to log a message, so using this mechanism to rotate logfiles must be used with caution since yxorp may not have closed the logfiles immediately upon reconfiguring).

Example

The following example will create three log files, and set the formatting for the access log and the error log:

```
<log accesslog=”/var/log/yxorpaccess” errorlog=”/var/log/yxorperror”
tracelog=”/var/log/yxorptrace”>
             <access-fmt>%clientip %nsend %method %host %uri %statuscode
</access-fmt>
             <error-fmt>%clientip %rejectreason”</error-fmt>
</log>
```

The following example, assuming a config snippet generated by a script running from cron, will “rotate” the access log file to a new, daily log file:

```
<log accesslog=”/var/log/yxorpaccess.2004-Feb-29” />
```

Configuring a configuration listener

The configuration listener is a special interface into an active Yxorp daemon. It allows the configuration of Yxorp to be changed while Yxorp is active.

Normally, the configuration listener should be bound to IP address 127.0.0.1 and port 7780. This is done by setting the <configlistener> tag in the configuration file. Setting the IP address to 127.0.0.1 is recommended, because of security reasons; only change this if you are sure you know all the consequences.

You can also choose not to run a configuration listener; this is inherently more secure, but also less flexible. If no configuration listener is active, the yxorpconfig command can not be used to reconfigure Yxorp, or to read the current configuration; also, all other commands like yxorpclientstate and yxorprealserver will not work without a configuration listener.

Disable the configuration listener by omitting the <configlistener> tag, or by explicitly setting the port number to zero.

There can only be one configlistener in an Yxorp daemon; any attempt to define more than one configlistener will overwrite the previous configuration attributes.

The <configlistener> tag takes the following attributes:

Attribute	Format	
port	number	port to bind to; default is 0. Setting the port number to 0 disables the configuration listener.
ipaddress	ipaddress	IPv4 address to listen on; default is 127.0.0.1

Reconfiguring

It is not possible to reconfigure the configlistener.

Example

The followin example shows how to configure the configuration listener to use IP address 127.0.0.1 and port number 7780:

```
<configlistener ipaddress=”127.0.0.1” port=”7780” />
```

Configuring threads

Yxorp uses a thread pool for processing the requests. The behaviour and size of the thread pool can be configured by the <thread> tag.

The <thread> tag takes the following attributes:

Attribute	*Format*	
initial	number	The number of threads that Yxorp will create at start
minfree	number	If less than this number of threads are free to process a request, Yxorp attempts to create a new thread.
max	number	Upper limit to the size of the thread pool.

Not all threads in an Yxorp process are counted for the thread pool size: the debug/log system, and each listener have their own dedicated thread that is not part of the thread pool.

Reconfiguring

It is possible to reconfigure threads; however, reducing 'max' will not lower the number of active threads.

Example

```
<threads initial=”20” minfree=”2” max=”100” />
```

Configuring security

Yxorp runs with reduced privileges on Solaris and Linux. Normally, binding to a port number lower than 1024 requires the process to run as root. Yxorp can release the root privileges and set parts of the process to run as a lower-privileged user. The way this works has significantly changed since version 0.19, it is now implemented using capabilities (Linux) and privileges (Solaris). For other platforms, you will either have to run Yxorp as root, or use non-privileged ports only.

For both Linux and Solaris, all privileges except the ones necessary for Yxorp are released. For Linux, the only remaining privilege is “cap_net_bind_service”; for Solaris, it is “PRIV_NET_PRIVADDR”.

Besides this, it is also possible to change the userid and group that Yxorp will run as; this has the added effect of securing access to the file system.

Another security setting is “chroot”; if this is set, yxorp will change root to the specified directory.

The <security> tag takes the following attributes:

Attribute	*Format*	
user	number	The numerical userid that will be set

Attribute	*Format*	
group	number	The numerical group that will be set
chroot	string	chroot to the specified directory just after reading the configuration file. The path must be complete (i.e. start in the root of the file system).

Reconfiguring

It is not possible to reconfigure security settings.

Example

```
<security user=”65535” group=”65535” />
```

Configuring basic authentication

Yxorp can store a table of basic authentication credentials in its configuration. Currently this is the only way, but it is foreseen that other tables may be available in a future version.

Note that basic authentication is not very secure. To recover userid and password from the encoded form is trivial. Don't use userid's and passwords you also use for other systems.

The <basicauth> tag takes the following attributes:

Attribute	*Format*	
realm	string	The realm. See also the first parameter on the basic_auth_check function.
userid	string	The user name
passwd	string	The password
base64	string	The encrypted form of userid and password (set either base64 or userid/passwd). Generated configurations from yxorp will always contain the base64 form.
remove	number	If nonzero, remove this base64 credential from the table.

Reconfiguring

It is possible to reconfigure all basic authentication settings.

Example

```
<basicauth realm=”mystuff” userid=”baas” passwd=”notsosecure” />
```

Configuring global settings

Global configuration settings are (as the name implies) global within a daemon (and thus govern all processing for all listeners).

The <globalconfiguration> tag takes the following attributes:

Attribute	*Format*	
maxchunksize	number	In chunked content processing, the maximum size of a chunk that will be accepted. A larger chunk will lead to the request being rejected.
maxserverconnectionattempts	number	The maximum number of times connection to a server will

Attribute	*Format*	
		be retried. If applicable, load balancing is applied, so the first connection attempt may be to server A, the next attempt to server B. Setting this to a higher number may lead to high response times if servers are unavailable. Also see connectservertimeout.
writetoclienttimeout	number	The time (in milliseconds) yxorp will wait for a write to a client to succeed. Note that normally a write will succeed almost instantaneously, since Unix processes the writes asynchronously; the actual meaning of this timer is more along the lines of "wait this long for buffer space to become available". If this timer expires, the request will be rejected.
readfromclienttimeout	number	The time (in milliseconds) yxorp will wait for data to arrive from a client. If the timer expires, the request will be rejected.
maxrealserverconnectfailuresbeforeoutservice	number	When this many subsequent failures occurred connecting a server, it is set out of service. If part of a virtualserver load balancing group, it will thereafter be considered out of service (and only be scheduled if the wakeup algorithm is active). For servers that are directly mapped to real servers, the out of service attribute is ignored.
connectservertimeout	number	(milliseconds) Wait this long for a connection to a server to complete.
writetoservertimeout	number	(milliseconds) Wait this long for a write to a server connection to succeed. See writetoclienttimeout above. If this timer expires, the request will be rejected.
readfromservertimeout	number	(milliseconds) Wait this long for data to arrive from a server connection. If this timer expires, the request will be rejected.
tunnelwritetoservertimeout	number	(milliseconds) Wait this long for a tunnel buffer to be written to the server connection.
tunnelwritetoclienttimeout	number	(milliseconds) Wait this long for a tunnel buffer to be written to the client connection.
tunnelconnecttimeout	number	(milliseconds) Wait this long for a tunnel connection to be established.
clientstatecleanupinterval	number	(seconds) Time between runs of the client state cleanup process. Don't set too short, because the cleanup process locks the client state table, and frequent locking decreases throughput. Also don't set too long, because then the amount of work increases and the table is locked longer.
clientstatecleanupmaxage	number	(seconds) Time a client state is maintained, counting from the last request referencing this state.
requestdvarvectorsize	number	The size of the dvar vector that is allocated for each request. This number should be set carefully, since the dvar table will not expand. The number should be large enough to accommodate all rule programs, built-in variables, and headers, but it should also be small enough not to waste

Attribute	*Format*	
		memory. Also consider that due to the way the dvar table works, a certain amount of free space increases the likelyhood that a dvar entry will be found quickly, whereas setting the vector size small will increase the likelyhood that a partial table scan will be necessary to locate a dvar entry. To improve efficiency, the number you set here will be rounded up to the next higher prime number. Note that if this number is set too small Yxorp will not be able to store information about the request; this may cause the request to be rejected. A suggested minimum is 67; when tuning, set to at least 2 times the actual dvars you need (counting built-ins, headers, and variables you use in rule programs, including results from capturing regexpses).
clientstatedvarvectorsize	number	The size of the dvar vector associated with a client state. In this dvar vector, the mapping between virtual and real servers is kept for sticky load balancing. The dvar vector can also be accessed from rule programs. To improve efficiency, the number you set here will be rounded up to the next higher prime number. Note that if this number is set too small Yxorp will not be able to store information in the client state; this may cause Yxorp to be unable to do sticky load balancing. A suggested minimum is 1; when tuning, set to at least 2 times the actual dvars you need (counting sticky-enabled virtualservers in the same cookie domain, and clientstate variables you use in rule programs).
pidfilename	string	The filename for the pid file, as a complete path starting in the root of the filesystem. Yxorp needs to have its own directory to put the pid file in, and the name of that directory must end in “yxorp”, otherwise, Yxorp will refuse to start. Default is “/var/run/yxorp/yxorp.pid”. Yxorp will attempt to create the last directory in the path if it does not exist. Also, Yxorp takes care of setting file ownerships to the user and group configured in the <security> tags.

The header table

Also in the globalconfiguration section is the table of accepted headers. Only headers that appear in this table are processed by yxorp, or made available in the rule language. The standard headers in RFC2616 are included by default. The header table can be changed by including “<header>” tags in the globalconfiguration section.

The <header> tag takes the following attributes:

Attribute	*Format*	
id	string	The name of the header, as it would appear in a request. Yxorp processes this value case-insensitively.
xlateid	string	The name of the header as it would appear in the rule language. Normally, the only difference between the id and xlateid are that '-' is

Attribute	Format	
		changed to '_'.
client	number	If non-zero, this header is accepted from a client connection. If zero, it is discarded, or the request is rejected according to the reject setting.
server	number	If non-zero, this header is accepted from a server connection. If zero, it is discarded, or the request is rejected according to the reject setting.
maxlen	number	The maximum length of this header line (so including the header name itself). If this length is exceeded, action is based on the reject setting for this header.
reject	number	If non-zero, this request is rejected. If zero, the header line not conforming to one of the specifications (currently client, server, maxlen and check) is discarded. Rejected headers are listed in the special variable 'rejectedheaders'.
check	string	Takes the values "none", "rfc2616-text", and "reduced"; this sets the character set that the header will be checked against. If characters exist in the header that are not in the character set, the header will either be discarded or the request will be rejected, according to the reject setting for the header. "none" means no checking will be performed; "rfc2616-text" sets the character set to the set described in RFC2616 (i.e. 7-bit ASCII without the control characters); "reduced" is a subset of rfc2616-text, in which most non-alphanumerical characters have been removed.

Reconfiguring

All global settings can be reconfigured. However, header entries, once they are created, may be modified but not completely removed (you can still set that the header will not be accepted though).

Example

The example below sets global configuration values, and updates the header table with the (custom) header "Example:", which will be called "bad_example:" in the scripting language, is accepted from a client but not from a server; the maximum length is 1 byte, and nonconformance will lead to the request being rejected.

```
<globalconfiguration
                    maxchunksize="16777216"
                    maxserverconnectionattempts="3"
                    writetoclienttimeout="15000"
                    readfromclienttimeout="15000"
                    maxrealserverconnectfailuresbeforeoutservice="4"
                    connectservertimeout="5000"
                    readfromservertimeout="15000"
                    writetoservertimeout="15000"
                    tunnelwritetoservertimeout="5000"
                    tunnelwritetoclienttimeout="5000"
                    tunnelconnecttimeout="5000"
                    clientstatecleanupinterval="30"
                    clientstatecleanupmaxage="900"
                    >
<header id=”Example”
        xlateid=”bad_example”
        maxlen=”1”
        client=”1”
        server=”0”
        reject=”1”
        check=”reduced”
/>

</globalconfiguration>
```

Rule language reference

Introduction

The rule language uses a style similar to C. It works differently though. The language has its own data types and statement constructions that are specifically tailored to dealing with requests and responses in a proxy context.

Some of the most obvious peculiarities when compared to a 'normal' programming language are:

- There is only one data type: a sequence of characters (a 'string').
- Literals must be enclosed in double quotes, unless each character is a number. There is, however, no difference in how the literal is stored: Numbers are represented in exactly the same way as strings. The handling of numbers is not especially strong; Yxorp was not designed to do number crunching.
- Truth values are represented by an empty string for false, and a non-empty string for true. The virtual machine and the built in functions tend to return true as a string containing "T".
- Variables are not declared; the parser will automatically declare variables 'on the fly'. In general, if a variable is referenced in any way, it is created. Reading a non-existent variable will result in an empty string.
- Variable names are case insensitive.
- Variable names may include a colon (':'). A variable name ending in a colon is used for variables containing header values; creating (i.e. referencing) a variable name ending in a colon causes Yxorp to create a HTTP header with the tag set to the variable name, and the value set to the variable's value.
- Variables at runtime comprise the actual value and a number of attributes; these determine the actual header name, the order of the header in the list of headers to be sent, etc.
- It is not possible to define functions. Function calls are available, but all called functions must be predefined, and their runtime must be included in the virtual machine.
- Since version 0.21 it is possible to do a function-call to another rule, but this functionality is very limited. It is not (yet) possible to pass parameters, or retrieve a return value.

Statements

The language has the usual constructions for expressions and if-then-else statements. There is only one statement that causes a loop: the 'foreach' statements, that takes the value of an expression to control the loop. Specific to the task of processing requests and responses are the whitelist and blacklist statements.

Variables

A variable consists of the following items:

- The variable name, as used in the program source
- The current value
- The 'original name'; this is the case-sensitive token used to generate the actual header name.

  The original name is not in any way dependent on the variable name; however, the variables created by Yxorp for the RFC2616-defined headers are almost the same as the original names associated to the variables (an exception is the '-' character, which is replaced by '_').
- The order number; this is used to specify the header order in a request or response. Yxorp sets the order number of the first header in a request to 10, incrementing by 10 for each next header.
- The variable's actual length.

- The variable's maximum length.
- A flag indicating if this variable has been truncated by the maximum length setting.
- A flag marking this variable as a request header, response header, or a header set for rejecting requests.
- A count field; in case multiple instances ('duplicate dvars' or 'dupvar') of a variable exist, the base variable will have it's count field set to the highest sequence number of the set of multiple instances.

  The first variable will have it's name set to the base name; subsequent variables will have a sequence number suffixed to the base name, where the sequence number starts at 1 for the first duplicate.

  Normally, these variables are only referenced through a foreach statement.

All of these variables and variable attributes may be referenced or modified in a rule program; most attributes can be accessed through function calls.

Expressions and operator precedence

Because the rule language is typeless, expressions must be written carefully. Since the only data type is a string, normal (numerical) comparison operators only work as intended if the data they operate on is exactly as expected. For instance,

```
i=”one”;
if (i==1) {
   ...
}
```

will produce unexpected results, because the variable i can not be converted to a number. The conversion will produce a default value (in the current versions most likely 0, but this may change in some future version). The numerical == operator will use the converted value, and return false.

Operators are grouped in precedence order in the usual way. Parentheses can be used to make the ordering explicit, or to change the order. The following code snippets give some examples of how this works:

```
i=2*3+1;            // i is set to 7
i=2*(3+1);          // i is set to 8
if (errorcode==302 && uri -/index\.html/) // evaluate errocode==302,
                                          // then evaluate uri -/index\.html/
                                          // then do logical and
```

The operator precedence is according to the following table:

Operator	***Description***	***Associativity***
!	logical not	right
-/.../	caseless regular expression	left
~/.../	casefull regular expression	left
*, /	numerical multiplication, division	left
+, -	numerical addition, subtraction	left
<, >, <=, >=	numerical comparison	left
!=, ==	numerical comparison	left
&&	logical and	left
||	logical or	left
=	assignment	left

Special variables

Some 'special' variables are used by Yxorp. These are:

Name	***Use***
uri	generating requests. The uri actually sent to the server will be taken from this variable. In the response stage, changing will have no effect but the changed value will be logged.
method	set to the method name. Use for reference only; changing this variable will have no effect.
rejectreason	holds the text message explaining the reason that the Yxorp base code rejected the request.
statuscode	holds the status code reported by the server that processed the request
errorcode	if set, the error code from this variable will be reported to the client instead of the default (400 – Bad Request) in case of a rejected request
errormessage	if set, the error reason string from this variable will be reported to the client instead of the default (400 – Bad Request) in case of a rejected request.
errorhtml	if set, html code contained in this variable will be inserted in the reject message.
errortitle	if set and errorhtml is not set, this variable defines the contents of the <title> tag in a reject page.
errorrejectreason	if set and errorhtml is not set, this variable allows an override of the default reason string in a reject page.
rejectedheaders	if Yxorp rejects a header (because it does not know this header, it is overlength, contains illegal characters) the header name is added to this variable.
_pattern	Set to the last executed regular expression pattern within the context of whitelist and blacklist statements. Not set for normal regular expressions.
_[0-9]	Set after the execution of a regular expression, depending on the pattern. Only the variables needed to capture the data extraction from the pattern are updated; other variables are untouched. Note that this functionality is only available if you have PCRE included in the build.

Rule syntax

This section is a reference of the rule program syntax.

Program

```
program: statements
```

A program is the entirety of the source code in a single rule. A program (i.e. rule) is comprised of one or more statements.

Note that a rule can not be empty; at least one statement must be in the rule, or the rule will fail to compile.

Block

```
block: { statements }
```

A block is used to group one or more statements. As in many other languages, this is most often used together with other statements, like the if statement.

Statements

```
statement: if (expression) statement
or         if (expression) statement else statement
or         foreach identifier (expression) statement
or         whitelist identifier { list-elements } if-failed statement
or         blacklist identifier { list-elements } if-failed statement
or         expression
or         block
or         return
```

There are several statements, as listed here. Note that a block can contain statements; by extension, a statement that contains another statement can also contain a block, and thus more statements.

If

```
if:        if (expression) statement
or         if (expression) statement else statement
```

The if statement works exactly as you would expect. Note that the expression is expected to result in a truth value. In rules, true is anything that is not an empty string; false is an empty string.

The simple if statement (without else) can also be written as follows:

```
if-shorthand: '?' expression statement
```

Foreach

```
foreach: foreach identifier (expression) statement
```

The foreach statement implements loops. The expression in the statement is expected to result in a space separated list. The foreach statement loops once for each element in the list, setting the variable named in the identifier to the list element.

The foreach statement is supported by several function calls, that deliver lists of variable names.

Whitelist

```
whitelist:       whitelist identifier { list-elements } if-failed statement

list-elements:   -/regexp/                    // simple, case insensitive
or               ~/regexp/                    // simple, case sensitive
or               -/regexp/ : list-statement   // complex, case insensitive
or               ~/regexp/ : list-statement   // complex, case sensitive

list-statement:  statement + continue-list
```

The whitelist statement is used to easily check the value of a variable against a set of regular expressions. If a match is found, the rest of the whitelist statement is skipped. If no match is found, the statement following the if-failed keyword is executed.

All forms of list-elements may be used in the same whitelist. There are no syntax restrictions to code formatting, but normal practice is to write each regular expression on its own line.

If a regular expression is in the complex form, the statement following the regular expression is executed before the normal whitelist action. In this context, a special statement is available with the continue-list keyword. If the continue-list statement is executed, the whitelist is resumed as if no match had occurred; this can be used to handle exception cases.

Normally, the if-failed clause is used to stop execution, reject a request, etc; however, note that this must be done explicitly, since the rule program language does not execute a default action.

After the whitelist completes, the special variable _pattern is set to the last executed pattern in a whitelist. This can be used to determine which pattern in the list matched. If whitelists are nested, the fact that _pattern contains the last executed pattern, not the successful match, may produce unexpected results.

Note that the scope in which _pattern is valid is different in a whitelist and a blacklist.

Blacklist

```
blacklist:        blacklist identifier { list-elements } if-failed statement

list-elements:    -/regexp/                    // simple, case insensitive
or                ~/regexp/                    // simple, case sensitive
or                -/regexp/ : list-statement   // complex, case insensitive
or                ~/regexp/ : list-statement   // complex, case sensitive

list-statement:   statement + continue-list
```

As with the whitelist statement, the blacklist statement is used to easily check the value of a variable against a set of regular expressions. If a match is found, the statement following the if-failed keyword is executed. If no match occurs, the if-failed clause is not executed.

All forms of list-elements may be used in the same blacklist. There are no syntax restrictions to code formatting, but normal practice is to write each regular expression on its own line.

If a regular expression is in the complex form, the statement following the regular expression is executed before the normal blacklist action. In this context, a special statement is available with the continue-list keyword. If the continue-list statement is executed, the blacklist resumed as if no match had occurred; this can be used to handle exception cases.

Normally, the if-failed clause is used to stop execution, reject a request, etc; however, note that this must be done explicitly, since the rule program language does not execute a default action.

After the blacklist executes, and within the context of the if-failed clause, the _pattern special variable contains the pattern that last executed. If blacklists are nested, the fact that _pattern contains the last executed pattern, not the successful match, may produced unexpected results.

Note that the scope in which _pattern is valid is different in a whitelist and a blacklist.

Return

```
return:    return
or         return (expression)
```

The return statement ends the execution of the current rule program. Either version can be used, but the value of the expression in a return statement is currently not used. This may change in future versions.

Expression

The forms that an expression may have are listed below:

```
expression:   call(parameters)           // function call
or            literal                    // value
or            number                     // integral number value
or            identifier = expression    // assignment
or            identifier                 // variable
or            $identifier                // indirect variable
or            ! expression               // logical not
or            expression && expression   // logical and
or            expression || expression   // logical or
or            expression + expression    // numerical addition
or            expression – expression    // numerical subtraction
or            expression * expression    // numerical multiplication
or            expression / expression    // numerical division
or            expression == expression   // numerical compare equal
or            expression != expression   // numerical compare not equal
or            expression < expression    // numerical compare less
or            expression > expression    // numerical compare greater
or            expression <= expression   // numerical compare less or equal
or            expression >= expression   // numerical compare greater or equal
or            (expression)               // order expression priority
or            identifier -/regexp/       // case insensitive regexp
or            identifier ~/regexp/       // case sensitive regexp
```

Most expression forms are very common, and will not be detailed further.

Note that the numerical operations are only valid if the operands are actually numbers.

Identifier

```
identifier: letter { letter | digit | "_" | ":" }
```

Identifiers are used for variable names. They must start with a letter, and may contain digits, underscores, and colons. Colons should only be used at the end, and signify that the identifier points to a variable that is used to store the contents of a header.

Internally, identifiers are also used for function names, and will be used in future versions to refer to other rules.

Numbers

```
number: digit { digit }
```

Numbers are formed as one or more digits. Decimal points (i.e. floating point numbers) are not supported. Negative numbers are also not supported.

In the internal representation, all numbers are stored as strings. There is no difference between 0 and "0" in the rule syntax.

Literals

```
literal: '"' { character } '"'
```

Literals are values that are expressed directly in the source, as in "index.html". Think of them as strings.

Indirect variable

```
indirect variable : $identifier
```

An indirect variable contains the name of another variable; this variable's value is the value of the expression form listed here. If the indirect variable is used without the indirection operator '$', the variable

itself is addressed; if the indirection operator is used, the value of the variable points to another variable.

Regular expressions

```
regular expression: identifier -/regexp/       // case insensitive regexp
or                  identifier ~/regexp/       // case sensitive regexp
```

Regular expressions are dependent on the regular expression library you included (libc or PCRE). With the posix-libc variant, only basic regexps are available; see your man page for regcomp, regexec, or regex for a description of what you can do if you have this library. Yxorp uses this variant only if PCRE could not be found on your system during configuration. PCRE is highly recommended, as it has much more functionality, and more consistent over different types of system. If you are not sure which library you have, check with yxorp -V how your build is configured.

If you included PCRE, you can use most constructs that are possible in Perl. Extraction of matched data from the original string is supported using the _0, _1, up to _9 special variables. Note that as in Perl, only the variables that were actually necessary for storing matched data are updated.

Both case-sensitive and case-insensitive variants of the regexp calls are available. Respectively use a tilde '~' or a minus '-' to specify which you want to use. The regexp itself must be enclosed in slashes. There is no implicit string begin or end added to the regexp; if you want to match against the start or end of a string, use $ and ^, respectively.

Function reference

basic_auth_check(realm, “local”) – enforce basic authentication

check if valid basic authentication credentials are present in the request (i.e. the Authorization: Basic header). If this is not found, the request is rejected with a 401 status code (normally causing a browser to show the userid/password dialog). The realm (text) is shown in this dialog window, and is also checked against the basic authentication credentials table.

If the authentication was successful, true is returned; false otherwise (and the request rejected). Note that execution of the rule program does not stop if the basic_auth_check fails; this must be done explicitly; normally, a return statement should be used to prevent execution of statements following the call to basic_auth_check. Typical use is as:

```
// check if basic authentication credentials are set
if (!basic_auth_check(“my-realm”, “local)) {
   return;          // end the rule, so that the implicit reject
                    // from basic_auth_check is processed
}
// reach here if basic_auth_check was successful
```

The second parameter to the basic_auth_check function *must* be exactly “local”. This refers to the use of an internal table for the authentication credentials. Future versions may support other tables; then, this parameter will be used to indicate which table should be used.

clientinrange(range) – check if client IP is in a range

clientinrange checks if the ip address that the client uses on this connection is part of the IPv4 range that is passed. If the client address or the range can not be parsed, false will be returned. Note that the client address can not be parsed if it is in IPv6 format, as would happen if the request comes in from an IPv6 listener. The range must be in the format a.b.c.d/x, where 1<=x<=32.

Typical use is as follows:

```
// check where the request comes from
if (clientinrange(127.0.0.1/32) {
   // allow some things
} else if (clientinrange(192.0.2.0/24) {
   // allow some other things
} else {
   // don't allow things
   reject("sorry...");
}
```

clientinip6range – check if client IP is in an IPv6 range

Similar to clientinrange, but for IPv6 addresses. The same limitations apply; IPv4 clients coming in through an IPv4 mode listener can not be correctly processed by this function.

The range can be specified in the following formats:

```
x:x:x:x:x:x:x:x/y            // default
x::x/y                       // missing parts are set to zeros
::a.b.c.d/y                  // deprecated transitional form
::ffff:a.b.c.d/y             // ipv4 mapped ipv6 address
::1/y                        // localhost

x = 4-digit hexadecimal
y = decimal, 1<=y<=128
a.b.c.d = ipv4 address range
```

clientstate(type) – retrieve value from client state

clientstate retrieves values from the client state, depending on the string passed to it:

"toclient": the number of bytes sent to the client on this clientstate (cumulative)

"toserver": the number of bytes sent to the server on this clientstate (cumulative)

"fromclient": the number of bytes received from the client on this clientstate (cumulative)

"fromserver": the number of bytes received from the server on this clientstate (cumulative)

"hitcount": the number of requests processed on this clientstate

"id": the cookie value for this clientstate

concat(...) – concatenate

concat takes a variable number of arguments and concatenates them. The concatenated string is returned.

contains(haystack, needle) – test if needle contains haystack

contains is equal to the C function strstr. It takes two arguments; the first is the haystack; the second is the needle. It returns a true if the needle is found in the haystack; false otherwise. The comparison is case sensitive.

Note you can also use a regular expression; this is a lot more flexible.

contains_characters(string, list) – test if string contains characters in list

contains_characters takes two arguments; the first is a string, which is tested against a list (which is also a string, by the way). If any of the characters in the list occur in the string, true is returned; false otherwise. The comparison between characters is case sensitive.

containscase(haystack, needle) – test if needle contains haystack

containscase is equal to the C function strcasestr (if that exists on your platform). It takes two arguments; the first is the haystack; the second is the needle. It returns true if the needle is found in the haystack; false otherwise. The comparison is case insensitive.

Note you can also use a regular expression; this is a lot more flexible.

enumerate_dupvar(variablename) – enumerate duplicate variables

enumerate_dupvar takes, as its only argument, a string containing the name of a variable. It returns a space separated string containing the real variable names of all variables in a duplicate variable set. If the variable is singular (i.e. there are no other members in the duplicate variable set) only the variable's name is returned.

enumerate_reqhdr() – enumerate request header variables

enumerate_reqhdr takes no arguments. It returns a space separated string containing the variable names of all variables that have the REQHDR attribute set. Yxorp sets this attribute for all variables it creates to represent request headers.

enumerate_rsphdr() – enumerate response header variables

enumerate_rsphdr takes no arguments. It returns a space separated string containing the variable names of all variables that have the RSPHDR attribute set. Yxorp sets this attribute for all variables it creates to represent response headers.

equal(s, r) – compare two variables

equal performs a case sensitive comparison of two variables.

equalcase(s, r) – case insensitive compare of two variables

equal performs a case insensitive comparison of two variables.

findheader(header) – retrieve contents of non-standard headers

findheader takes one parameter: the exact, case-sensitive name of a header. It returns the value of this header in the current request or response, or false if it is not found. findheader may only be used in request or response rules; the results are undefined for other rule types. The header is not processed in any way, and the settings in the <header> tags in the <globalconfiguration> section do not apply.

getattr(variablename, attr) – test attributes of a variable

getattr tests a variable for a specific attribute. The variable name must be set as the first parameter; the exact attribute as the second. If the attribute is present, true will be returned; if not, false will be returned.

getclientip() – get the client ip address

getclientip returns the ip address (IPv4 or IPv6) of the client associated with the current request.

getclientdomainname() – get the client domain name

getclientip returns the domain name of the client associated with the current request. Note that using this function causes a domain name lookup to be done; this may impact performance, especially for large volumes.

getclientstatedvar(variablename) – read a variable from the client state

getclientstatedvar reads a variable from the client state, if one exists for this request. The variable name must be set as the first parameter. If the variable and the client state exist, the data will be returned; otherwise, an empty string (i.e. false) will be returned.

getlength(variablename) – returns the length of a variable

getlength returns the length of a variable. The name of the variable is passed to getlength.

getlistenerid() – returns the listener id

getlistenerid returns the id (i.e. name) of the listener that has received the request that is currently being processed.

getmaxcount(variablename) – return the number of variables in a dupvar group

getmaxcount returns the number of variables in a duplicate variable group

getmaxlength(variablename) – return the maxlength attribute of a variable

getmaxlength returns the maxlength attribute of a variable.

getorder(variablename) – return the order attribute of a variable

getorder returns the order attribute of a variable.

getoriginalname(variablename) – return the originalname attribute of a variable

getorder returns the originalname attribute of a variable.

getserverturnaround() – return the turnaround time for the server request

getserverturnaround returns the time, in milliseconds, that it took from sending the request to the server, to the response header being completely received. This gives an indication of server response; note that this time is however in some cases quite different from the response time an end user may experience.

getsslbackend() – return the sslbackend flag

getsslbackend returns the value of the sslbackend flag. If set, this means Yxorp will use SSL to the server; if not set, Yxorp will use plain HTTP.

getsticky() – return the sticky flag

getsticky returns the value of the sticky flag.

issslsession() – return the SSL state of the client session

issslsession returns true if the client session uses SSL; false otherwise.

redirect(url) – redirect request

redirects the request, by rejecting it with a statuscode 307 (Temporary Redirect), passing the URL in a Location: header. This causes a browser to redirect to this URL.

reject(reason) – reject request

the request is rejected, with the specified reason. The reason string is also logged to the error log.

setattr(variablename, type) – set variable attribute

Sets the attribute on the specified variable. The following attribute values can be set:

RSPHDR: this variable is part of the response header group.

REQHDR: this variable is part of the request header group.

REJHDR: this variable is part of the reject header group.

setclientstatedvar(variablename, value) – set a variable in the client state

setclientstatedvar sets a variable from the client state, if one exists for this request. The variable name must be set as the first parameter; its value as the second. Note that client states are created after a request type rule runs, so on the first request from a client, the client state is not yet available. Also note that the number of slots in the client state dvar table is normally very limited (but it can be increased in the global configuration).

setcookiedomain(string) – set cookie domain

This function sets the domain part of the state cookie that Yxorp uses for tracking or sticky load balancing. If no domain is set, the generally accepted behavior of clients is to only send back the cookie values to the same domain that set the cookie. If you use the domain part, you can cause the cookie values to be sent to several hostnames in a domain group (i.e. set the domain to .example.org to have the cookie sent to www.example.org, www2.example.org, etc).

setdup(variablename, value) – set duplicate variable

Sets the next instance of the duplicate variable to the specified value.

If, for example, a dupvar exists:

```
ex=an
ex1=example
```

the call *setdup("ex", "test")* would have the following result:

```
ex=an
ex1=example
ex2=test
```

setmaxlength(variablename, length) – set variable max length attribute

Sets the maximum length attribute on the specified variable (after this, setting a longer value in a variable results in the value being truncated).

setorder(variablename, ordinal) – set variable order attribute

Sets the order attribute on the specified variable.

setoriginalname(variablename, originalname) – set variable originalname attribute

Sets the originalname attribute on the specified variable.

setsslbackend() – set ssl backend

Forces the server session for this request to use SSL.

setsticky() – set sticky flag

Sets the sticky flag for this request, causing sticky scheduling and client tracking for this request.

statuscodemsg(code) – returns status code textual message

The text message for the HTTP response code is returned.

trace(message) – send a message to the trace log

The message is written to the trace log. If no trace log is active, the message is discarded.

unsetsslbackend() – clear ssl backend

Stops forcing the server session for this request to use SSL.

unsetsticky() – clear sticky flag

Clears the sticky flag for this request, disabling sticky scheduling and client tracking for this request.

yesno(x) – return truth value as yes or no

yesno takes the truth value from the argument and returns yes (=true) or no(=false).

The virtual machine

The code generated from the rule program sources is compiled, and executed by a virtual machine. The virtual machine implements a stack machine (like a RPN desk calculator) working on strings.

In general, the virtual machine is surprisingly fast. The instructions that the virtual machine processes are mostly very simple and straightforward. Even though the rule compiler is simple, the execution speed of the rule programs is not really impacted by the lack of optimization of the compiler. Most data manipulations are accomplished just by moving pointers around. Temporary variables are allocated in scratch memory pools; this speeds up execution because the memory allocation is much faster in this way, and also this ensures that all the used pointers will be valid for the duration of the VM run, even if memory shortages occur.

There is, of course, also a weak point: the virtual machine is not especially good at arithmetic. Since all data is represented as strings, calculations require several conversions. Number handling is also quite simplistic; if a string value is used in a numeric operation, the value will default to 0 instead of causing an error (this behavior is likely to change in a future version).

The use of memory pools causes that there is a limit to the maximum size of an object. Currently, this is in the order of 4K bytes, which should not be a limitation for typical use. If your application needs larger objects, source-level tuning is necessary.

As all other Yxorp modules, the virtual machine contains a large amount of debugging hooks. If you build Yxorp with debugging enabled, performance will be significantly lowered.